MindSpore API Documentation

Release r2.6.0

MindSpore

Jul 01, 2025

CONTENTS

CHAPTER

ONE

MINDSPORE

1.1 Data Presentation

1.1.1 Tensor

mindspore.Tensor	Tensor is a data structure that stores an n-dimensional array.
mindspore.tensor	Create a new Tensor in Cell.construct() or function decorated by @jit.
mindspore.COOTensor	A sparse representation of a set of nonzero elements from a tensor at given indices, where the indices indicate the position of each non-zero element.
mindspore.CSRTensor	Constructs a sparse tensor in CSR (Compressed Sparse Row) format, with specified values indicated by *values* and row and column positions indicated by *indptr* and *indices*.
mindspore.RowTensor	A sparse representation of a set of tensor slices at given indices.
mindspore.SparseTensor	A sparse representation of a set of nonzero elements from a tensor at given indices.
mindspore.is_tensor	Check whether the input object is *mindspore.Tensor* .
mindspore.from_numpy	Convert numpy array to Tensor.

mindspore.Tensor

class mindspore.**Tensor**(*input_data=None*, *dtype=None*, *shape=None*, *init=None*, *const_arg=False*, *device=None*)

Tensor is a data structure that stores an n-dimensional array.

Note:

- If *init* interface is used to initialize *Tensor*, the *Tensor.init_data* API needs to be called to load the actual data to *Tensor*.
- All modes of CPU and GPU, and Atlas training series with graph mode (mode=mindspore.GRAPH_MODE) do not supported in-place operations yet.
- The default value `None` of *input_data* works as a placeholder, it does not mean that we can create a NoneType Tensor.
- Tensor with *shape* contains 0 is not fully tested and supported.

Warning: To convert dtype of a *Tensor*, it is recommended to use *Tensor.astype()* rather than *Tensor(sourceTensor, dtype=newDtype)*.

Parameters

- **input_data** (Union[Tensor, float, int, bool, tuple, list, numpy.ndarray]) - The data to be stored. It can be another Tensor, Python number or NumPy ndarray. Default: `None` .
- **dtype** (*mindspore.dtype*) - Used to indicate the data type of the output Tensor. The argument should be defined in *mindspore.dtype*. If it is `None` , the data type of the output Tensor will be the same as the *input_data*. Default: `None` .
- **shape** (Union[tuple, list, int, *mindspore.Symbol*]) - Used to indicate the shape of the output Tensor. If *input_data* is available, *shape* doesn't need to be set. If `None` or *Symbol* exists in *shape* , a tensor of dynamic shape is created, *input_data* doesn't need to be set; if only integers exist in *shape*, a tensor of static shape is created, *input_data* or *init* must be set. Default: `None` .
- **init** (Initializer) - The information of init data. *init* is used for delayed initialization in parallel mode, when using init, *dtype* and *shape* must be set. Default: `None` .
- **const_arg** (bool) - Whether the tensor is a constant when it is used for the argument of a network. Default: `False` .
- **device** (str) - This parameter is reserved and does not need to be configured. Default: `None` .

Outputs:

Tensor.

Examples:

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.common.initializer import One
>>> # initialize a tensor with numpy.ndarray
>>> t1 = Tensor(np.zeros([1, 2, 3]), ms.float32)
>>> print(t1)
[[[0. 0. 0.]
[0. 0. 0.]]]
>>> print(type(t1))
<class 'mindspore.common.tensor.Tensor'>
>>> print(t1.shape)
(1, 2, 3)
>>> print(t1.dtype)
Float32
>>>
>>> # initialize a tensor with a float scalar
>>> t2 = Tensor(0.1)
>>> print(t2)
0.1
>>> print(type(t2))
<class 'mindspore.common.tensor.Tensor'>
>>> print(t2.shape)
()
>>> print(t2.dtype)
Float32
>>>
>>> # initialize a tensor with a tuple
>>> t3 = Tensor((1, 2))
>>> print(t3)
[1 2]
>>> print(type(t3))
<class 'mindspore.common.tensor.Tensor'>
```

(continues on next page)

(continued from previous page)

```
>>> print(t3.shape)
(2,)
>>> print(t3.dtype)
Int64
...
>>> # initialize a tensor with init
>>> t4 = Tensor(shape = (1, 3), dtype=ms.float32, init=One())
>>> print(t4)
[[1. 1. 1.]]
>>> print(type(t4))
<class 'mindspore.common.tensor.Tensor'>
>>> print(t4.shape)
(1, 3)
>>> print(t4.dtype)
Float32
```

mindspore.Tensor.abs	For details, please refer to *mindspore.ops.abs()*.
mindspore.Tensor.absolute	Alias for `Tensor.abs()`.
mindspore.Tensor.acos	For details, please refer to *mindspore.ops.acos()*.
mindspore.Tensor.acosh	For details, please refer to *mindspore.ops.acosh()*.
*mindspore.Tensor.add_*	In-place version of *mindspore.Tensor.add()*.
mindspore.Tensor.add	Adds other value to *self* element-wise.
mindspore.Tensor.addbmm	For details, please refer to *mindspore.ops.addbmm()*.
mindspore.Tensor.addcdiv	For details, please refer to *mindspore.ops.addcdiv()*.
mindspore.Tensor.addcmul	For details, please refer to *mindspore.ops.addcmul()*.
mindspore.Tensor.addmm	For details, please refer to *mindspore.ops.addmm()*.
*mindspore.Tensor.addmm_*	In-place version of *mindspore.Tensor.addmm()*.
mindspore.Tensor.addmv	For details, please refer to *mindspore.ops.addmv()*.
mindspore.Tensor.addr	For details, please refer to *mindspore.ops.addr()*.
mindspore.Tensor.adjoint	For details, please refer to *mindspore.ops.adjoint()*.
mindspore.Tensor.all	Tests if all element in tensor evaluates to *True* along the given axes.
mindspore.Tensor.allclose	Returns a new Tensor with boolean elements representing if each element of *self* is "close" to the corresponding element of *other*.
mindspore.Tensor.amax	For details, please refer to *mindspore.ops.amax()*.
mindspore.Tensor.amin	For details, please refer to *mindspore.ops.amin()*.
mindspore.Tensor.aminmax	For details, please refer to *mindspore.ops.aminmax()*.
mindspore.Tensor.any	Tests if any element in tensor evaluates to *True* along the given axes.
mindspore.Tensor.angle	For details, please refer to *mindspore.ops.angle()*.
mindspore.Tensor.approximate_equal	For details, please refer to *mindspore.ops.approximate_equal()*, The parameter *other* of current interface is the same as the parameter *y* of the reference interface.
mindspore.Tensor.arccos	Alias for *mindspore.Tensor.acos()*.
mindspore.Tensor.arccosh	Alias for *mindspore.Tensor.acosh()*.
mindspore.Tensor.arcsin	Alias for *mindspore.Tensor.asin()*.
mindspore.Tensor.arcsinh	Alias for *mindspore.Tensor.asinh()*.
mindspore.Tensor.arctan	Alias for *mindspore.Tensor.atan()*.
mindspore.Tensor.arctan2	Alias for `Tensor.atan2()`.
mindspore.Tensor.arctanh	Alias for *mindspore.Tensor.atanh()*.

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.argmax	Return the indices of the maximum values along the given axis of the tensor.
mindspore.Tensor.argmax_with_value	Return the maximum values and their indices along the given axis of the tensor.
mindspore.Tensor.argmin	Returns the indices of the minimum values along the given axis of the tensor.
mindspore.Tensor.argmin_with_value	Return the minimum values and their indices along the given axis of the tensor.
mindspore.Tensor.argsort	Sorts *self* along the given dimension in specified order and return the sorted indices.
mindspore.Tensor.argwhere	For details, please refer to mindspore.ops.argwhere().
mindspore.Tensor.asin	For details, please refer to mindspore.ops.asin().
mindspore.Tensor.asinh	For details, please refer to mindspore.ops.asinh().
mindspore.Tensor.asnumpy	Convert tensor to numpy array.
mindspore.Tensor.assign_value	Assign another tensor value to this tensor.
mindspore.Tensor.astype	Return a copy of the tensor, cast to a specified type.
mindspore.Tensor.atan	For details, please refer to mindspore.ops.atan().
mindspore.Tensor.atan2	For details, please refer to mindspore.ops.atan2().
mindspore.Tensor.atanh	For details, please refer to mindspore.ops.atanh().
mindspore.Tensor.baddbmm	For details, please refer to mindspore.ops.baddbmm().
mindspore.Tensor.bernoulli	For details, please refer to mindspore.mint.bernoulli().
mindspore.Tensor.bfloat16	Converts input tensor dtype to *bfloat16*.
mindspore.Tensor.bincount	For details, please refer to mindspore.ops.bincount().
mindspore.Tensor.bitwise_and	Returns bitwise *and* of two tensors element-wise.
mindspore.Tensor.bitwise_not	Returns bitwise *not* of *self*.
mindspore.Tensor.bitwise_left_shift	For details, please refer to mindspore.ops.bitwise_left_shift().
mindspore.Tensor.bitwise_or	Returns bitwise *or* of two tensors element-wise.
mindspore.Tensor.bitwise_right_shift	For details, please refer to mindspore.ops.bitwise_right_shift().
mindspore.Tensor.bitwise_xor	Returns bitwise *xor* of two tensors element-wise.
mindspore.Tensor.bmm	For details, please refer to mindspore.ops.bmm().
mindspore.Tensor.bool	Converts input tensor dtype to *bool*.
mindspore.Tensor.broadcast_to	For details, please refer to mindspore.ops.broadcast_to().
mindspore.Tensor.byte	Converts input tensor dtype to *uint8*.
mindspore.Tensor.cauchy	Fills the tensor with numbers drawn from the Cauchy distribution.
mindspore.Tensor.ceil	For details, please refer to mindspore.ops.ceil().
mindspore.Tensor.cholesky	For details, please refer to mindspore.ops.cholesky().
mindspore.Tensor.cholesky_solve	For details, please refer to mindspore.ops.cholesky_solve().
mindspore.Tensor.choose	Construct a tensor from an index tensor and a list of tensors to choose from.
mindspore.Tensor.chunk	Cut the self Tensor into *chunks* sub-tensors along the specified dimension.
mindspore.Tensor.clamp	For details, please refer to mindspore.ops.clamp().
mindspore.Tensor.clamp_	In-place version of mindspore.Tensor.clamp().
mindspore.Tensor.clip	Alias for mindspore.Tensor.clamp().
mindspore.Tensor.clone	Returns a copy of self.
mindspore.Tensor.col2im	For details, please refer to mindspore.ops.col2im().

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.conj	For details, please refer to mindspore.ops.conj().
mindspore.Tensor.contiguous	Converts a Tensor into a continuous-memory Tensor that contains the same data as the original Tensor.
mindspore.Tensor.copy	Return a copy of the tensor.
mindspore.Tensor.copy_	Copies the elements from *src* into *self* tensor and returns *self*.
mindspore.Tensor.copysign	For details, please refer to mindspore.ops.copysign().
mindspore.Tensor.cos	For details, please refer to mindspore.ops.cos().
mindspore.Tensor.cosh	For details, please refer to mindspore.ops.cosh().
mindspore.Tensor.count_nonzero	Counts the number of non-zero values in the tensor input along the given dim.
mindspore.Tensor.cov	For details, please refer to mindspore.ops.cov().
mindspore.Tensor.cross	For details, please refer to mindspore.ops.cross().
mindspore.Tensor.cummax	For details, please refer to mindspore.ops.cummax().
mindspore.Tensor.cummin	For details, please refer to mindspore.ops.cummin().
mindspore.Tensor.cumprod	For details, please refer to mindspore.ops.cumprod().
mindspore.Tensor.cumsum	Computes the cumulative sum of self Tensor along *dim*.
mindspore.Tensor.deg2rad	For details, please refer to mindspore.ops.deg2rad().
mindspore.Tensor.diag	For details, please refer to mindspore.ops.diag().
mindspore.Tensor.diagflat	For details, please refer to mindspore.ops.diagflat().
mindspore.Tensor.diagonal	For details, please refer to mindspore.ops.diagonal().
mindspore.Tensor.diagonal_scatter	For details, please refer to mindspore.ops.diagonal_scatter().
mindspore.Tensor.diff	For details, please refer to mindspore.ops.diff().
mindspore.Tensor.digamma	For details, please refer to mindspore.ops.digamma().
mindspore.Tensor.div	For details, please refer to mindspore.ops.div().
mindspore.Tensor.div_	In-place version of mindspore.Tensor.div().
mindspore.Tensor.divide	Alias for mindspore.Tensor.div().
mindspore.Tensor.dot	Computes the dot product of two 1D tensor.
mindspore.Tensor.double	Converts input tensor dtype to *float64*.
mindspore.Tensor.dsplit	For details, please refer to mindspore.ops.dsplit().
mindspore.Tensor.dtype	Return the dtype of the tensor (mindspore.dtype).
mindspore.Tensor.eigvals	For details, please refer to mindspore.ops.eigvals().
mindspore.Tensor.eq	For details, please refer to mindspore.ops.eq().
mindspore.Tensor.equal	For details, please refer to mindspore.ops.equal().
mindspore.Tensor.erf	For details, please refer to mindspore.ops.erf().
mindspore.Tensor.erfc	For details, please refer to mindspore.ops.erfc().
mindspore.Tensor.erfinv	For details, please refer to mindspore.ops.erfinv().
mindspore.Tensor.exp	For details, please refer to mindspore.ops.exp().
mindspore.Tensor.exp_	Inplace version of mindspore.Tensor.exp().
mindspore.Tensor.expand	For details, please refer to mindspore.ops.broadcast_to().
mindspore.Tensor.expand_as	Expand the shape of the input tensor to be the same as the another input tensor.
mindspore.Tensor.expand_dims	For details, please refer to mindspore.ops.expand_dims().
mindspore.Tensor.expm1	For details, please refer to mindspore.ops.expm1().
mindspore.Tensor.exponential_	Fills *self* tensor with elements drawn from the exponential distribution:
mindspore.Tensor.fill_	Fills *self* tensor with the specified *value* .
mindspore.Tensor.fill_diagonal	Fills the main diagonal of a Tensor with a specified value and returns the result.

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.flatten	Flatten a tensor along dimensions from *start_dim* to *end_dim*.
mindspore.Tensor.flip	For details, please refer to mindspore.ops.flip().
mindspore.Tensor.fliplr	For details, please refer to mindspore.ops.fliplr().
mindspore.Tensor.flipud	For details, please refer to mindspore.ops.flipud().
mindspore.Tensor.float	Converts input tensor dtype to *float32*.
mindspore.Tensor.float_power	For details, please refer to mindspore.ops.float_power().
mindspore.Tensor.floor	For details, please refer to mindspore.ops.floor().
mindspore.Tensor.floor_	In-place version of mindspore.Tensor.floor().
mindspore.Tensor.floor_divide	Divides the self tensor by the other input tensor element-wise and round down to the closest integer.
mindspore.Tensor.floor_divide_	Divides the self tensor by the other tensor element-wise and round down to the closest integer.
mindspore.Tensor.flush_from_cache	Flush cache data to host if tensor is cache enable.
mindspore.Tensor.fmax	For details, please refer to mindspore.ops.fmax().
mindspore.Tensor.fmod	For details, please refer to mindspore.ops.fmod().
mindspore.Tensor.fold	For details, please refer to mindspore.ops.fold().
mindspore.Tensor.frac	For details, please refer to mindspore.ops.frac().
mindspore.Tensor.from_numpy	Convert numpy array to Tensor.
mindspore.Tensor.gather	Gather data from a tensor by indices.
mindspore.Tensor.gather_elements	For details, please refer to mindspore.ops.gather_elements().
mindspore.Tensor.gather_nd	For details, please refer to mindspore.ops.gather_nd().
mindspore.Tensor.gcd	For details, please refer to mindspore.ops.gcd().
mindspore.Tensor.ge	Alias for mindspore.Tensor.greater_equal().
mindspore.Tensor.geqrf	For details, please refer to mindspore.ops.geqrf().
mindspore.Tensor.ger	For details, please refer to mindspore.ops.ger().
mindspore.Tensor.greater	For details, please refer to mindspore.ops.greater().
mindspore.Tensor.greater_equal	For details, please refer to mindspore.ops.greater_equal().
mindspore.Tensor.gt	For details, please refer to :func:'mindspore.Tensor.greater'.
mindspore.Tensor.H	Returns a view of a matrix (2-D tensor) conjugated and transposed.
mindspore.Tensor.half	Converts input tensor dtype to *float16*.
mindspore.Tensor.hardshrink	For details, please refer to mindspore.ops.hardshrink().
mindspore.Tensor.has_init	Whether tensor is initialized.
mindspore.Tensor.heaviside	For details, please refer to mindspore.ops.heaviside().
mindspore.Tensor.histc	For details, please refer to mindspore.ops.histc().
mindspore.Tensor.hsplit	For details, please refer to mindspore.ops.hsplit().
mindspore.Tensor.hypot	For details, please refer to mindspore.ops.hypot().
mindspore.Tensor.i0	For details, please refer to mindspore.ops.bessel_i0().
mindspore.Tensor.igamma	For details, please refer to mindspore.ops.igamma().
mindspore.Tensor.igammac	For details, please refer to mindspore.ops.igammac().
mindspore.Tensor.imag	For details, please refer to mindspore.ops.imag().
mindspore.Tensor.index_add	Adds tensor *y* to specified axis and indices of tensor *self*.
mindspore.Tensor.index_add_	Accumulate the elements of *alpha* times *source* into the *self* by adding to the index in the order given in *index*.

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.index_fill	For details, please refer to mindspore.ops.index_fill().
mindspore.Tensor.index_put	Based on the indices in *indices*, replace the corresponding elements in Tensor *self* with the values in *values*.
mindspore.Tensor.index_put_	Based on the indices in *indices*, replace the corresponding elements in Tensor *self* with the values in *values*.
mindspore.Tensor.index_select	Generates a new Tensor that accesses the values of *self* along the specified *axis* dimension using the indices specified in *index*.
mindspore.Tensor.init_data	Get the tensor format data of this Tensor.
mindspore.Tensor.inner	For details, please refer to mindspore.ops.inner().
mindspore.Tensor.inplace_update	For details, please refer to mindspore.ops.inplace_update().
mindspore.Tensor.int	Converts input tensor dtype to *int32*.
mindspore.Tensor.inv	For details, please refer to mindspore.ops.inv().
mindspore.Tensor.inverse	For details, please refer to mindspore.ops.inverse().
mindspore.Tensor.invert	For details, please refer to mindspore.ops.invert().
mindspore.Tensor.isclose	Returns a tensor of Boolean values indicating whether each element of *input* is "close" to the corresponding element of *other*.
mindspore.Tensor.isfinite	For details, please refer to mindspore.ops.isfinite().
mindspore.Tensor.is_complex	For details, please refer to: func:'mindspore.ops.is_complex'.
mindspore.Tensor.is_contiguous	Determines whether the memory of tensor is contiguous.
mindspore.Tensor.is_floating_point	For details, please refer to mindspore.ops.is_floating_point().
mindspore.Tensor.isinf	For details, please refer to mindspore.ops.isinf().
mindspore.Tensor.isnan	For details, please refer to mindspore.ops.ne().
mindspore.Tensor.isneginf	For details, please refer to mindspore.ops.isneginf().
mindspore.Tensor.isposinf	For details, please refer to mindspore.ops.isposinf().
mindspore.Tensor.isreal	For details, please refer to mindspore.ops.isreal().
mindspore.Tensor.is_signed	Judge whether the data type of tensor is a signed data type.
mindspore.Tensor.item	Return the value of this tensor as standard Python number.
mindspore.Tensor.itemset	Insert scalar into a tensor (scalar is cast to tensor's dtype, if possible).
mindspore.Tensor.itemsize	Return the length of one tensor element in bytes.
mindspore.Tensor.lcm	For details, please refer to mindspore.ops.lcm().
mindspore.Tensor.ldexp	For details, please refer to mindspore.ops.ldexp().
mindspore.Tensor.le	For details, please refer to mindspore.ops.le().
mindspore.Tensor.lerp	For more details, please refer to mindspore.ops.lerp().
mindspore.Tensor.less	For details, please refer to mindspore.ops.less().
mindspore.Tensor.less_equal	For details, please refer to mindspore.ops.less_equal().
mindspore.Tensor.log	For details, please refer to mindspore.ops.log().
mindspore.Tensor.log10	For details, please refer to mindspore.ops.log10().
mindspore.Tensor.log1p	For details, please refer to mindspore.ops.log1p().
mindspore.Tensor.log2	For details, please refer to mindspore.ops.log2().
mindspore.Tensor.logaddexp	For details, please refer to mindspore.ops.logaddexp().
mindspore.Tensor.logaddexp2	For details, please refer to mindspore.ops.logaddexp2().
mindspore.Tensor.logcumsumexp	For details, please refer to mindspore.ops.logcumsumexp().
mindspore.Tensor.logdet	For details, please refer to mindspore.ops.logdet().

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.logical_and	For details, please refer to mindspore.ops.logical_and().
mindspore.Tensor.logical_not	For details, please refer to mindspore.ops.logical_not().
mindspore.Tensor.logical_or	For details, please refer to mindspore.ops.logical_or().
mindspore.Tensor.logical_xor	Computes the "logical XOR" of two tensors element-wise.
mindspore.Tensor.logit	For details, please refer to mindspore.ops.logit().
mindspore.Tensor.logsumexp	For details, please refer to mindspore.ops.logsumexp().
mindspore.Tensor.log_normal	Fills the elements of the input tensor with log normal values initialized by given mean and std:
mindspore.Tensor.long	Converts input tensor dtype to *int64*.
mindspore.Tensor.lt	For more details, please refer to mindspore.Tensor.less().
mindspore.Tensor.lu_solve	For details, please refer to mindspore.ops.lu_solve().
mindspore.Tensor.masked_fill	For details, please refer to mindspore.ops.masked_fill().
mindspore.Tensor.masked_fill_	In-place version of mindspore.Tensor.masked_fill().
mindspore.Tensor.masked_scatter	Updates the value in the "self Tensor" with the *tensor* value according to the mask, and returns a Tensor.
mindspore.Tensor.masked_select	For details, please refer to mindspore.ops.masked_select().
mindspore.Tensor.matmul	Returns the matrix product of two tensors.
mindspore.Tensor.max	Returns the maximum value of the self tensor.
mindspore.Tensor.maximum	For details, please refer to mindspore.ops.maximum().
mindspore.Tensor.mean	Reduces all dimension of a tensor by averaging all elements in the dimension, by default.
mindspore.Tensor.median	Computes the median and indices of input tensor.
mindspore.Tensor.t	Transpose *self* .
mindspore.Tensor.mH	Accessing this property is equivalent to Calling self.adjoint().
mindspore.Tensor.min	Returns the minimum value of the self tensor.
mindspore.Tensor.minimum	For details, please refer to mindspore.ops.minimum().
mindspore.Tensor.mm	Returns the matrix product of two arrays.
mindspore.Tensor.moveaxis	For details, please refer to mindspore.ops.moveaxis().
mindspore.Tensor.movedim	For details, please refer to mindspore.ops.movedim().
mindspore.Tensor.move_to	Copy Tensor to target device synchronously or asynchronously, default synchronously.
mindspore.Tensor.msort	For details, please refer to mindspore.ops.msort().
mindspore.Tensor.mT	Returns the Tensor that exchanges the last two dimensions.
mindspore.Tensor.mul	For details, please refer to mindspore.ops.mul().
mindspore.Tensor.mul_	Multiplies two tensors element-wise.
mindspore.Tensor.multinomial	For details, please refer to mindspore.ops.multinomial().
mindspore.Tensor.multiply	For details, please refer to mindspore.ops.mul().
mindspore.Tensor.mvlgamma	For details, please refer to mindspore.ops.mvlgamma().
mindspore.Tensor.nan_to_num	For details, please refer to mindspore.ops.nan_to_num().
mindspore.Tensor.nanmean	For details, please refer to mindspore.ops.nanmean().

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.nanmedian	For details, please refer to mindspore.ops.nanmedian().
mindspore.Tensor.nansum	Computes sum of input Tensor over a given dimension, treating NaNs as zero.
mindspore.Tensor.narrow	Obtains a tensor of a specified length at a specified start position along a specified axis.
mindspore.Tensor.nbytes	Return the total number of bytes taken by the tensor.
mindspore.Tensor.ndim	Return the number of tensor dimensions.
mindspore.Tensor.ndimension	Alias for mindspore.Tensor.ndim.
mindspore.Tensor.ne	Alias for mindspore.Tensor.not_equal().
mindspore.Tensor.neg	For details, please refer to mindspore.ops.neg().
mindspore.Tensor.negative	Alias for mindspore.Tensor.neg().
mindspore.Tensor.nelement	Alias for mindspore.Tensor.numel().
mindspore.Tensor.new_ones	Return a tensor of *size* filled with ones.
mindspore.Tensor.new_zeros	Return a tensor of *size* filled with zeros.
mindspore.Tensor.nextafter	For details, please refer to mindspore.ops.nextafter().
mindspore.Tensor.nonzero	Return the positions of all non-zero values.
mindspore.Tensor.norm	For details, please refer to mindspore.ops.norm().
mindspore.Tensor.normal_	Update the *self* tensor in place by generating random numbers sampled from the normal distribution which constructed by the parameters *mean* and *std*.
mindspore.Tensor.not_equal	For details, please refer to mindspore.ops.ne().
mindspore.Tensor.numel	For details, please refer to mindspore.ops.numel().
mindspore.Tensor.numpy	Alias for mindspore.Tensor.asnumpy().
mindspore.Tensor.orgqr	For details, please refer to mindspore.ops.orgqr().
mindspore.Tensor.ormqr	For details, please refer to mindspore.ops.ormqr(), Args *input2* and *input3* correspond to the args *tau* and *other* of mindspore.ops.ormqr().
mindspore.Tensor.outer	For details, please refer to mindspore.ops.outer().
mindspore.Tensor.permute	Tensor.permute supports unpacking the *axis* argument automatically when it is passed as an indefinite number of positional arguments, which has a slight difference from the input parameter of mindspore.ops.permute().
mindspore.Tensor.positive	For details, please refer to mindspore.ops.positive().
mindspore.Tensor.pow	For details, please refer to mindspore.ops.pow().
mindspore.Tensor.prod	Reduces a dimension of a tensor by multiplying all elements in the dimension, by default.
mindspore.Tensor.ptp	The name of the function comes from the acronym for "peak to peak".
mindspore.Tensor.rad2deg	For details, please refer to mindspore.ops.rad2deg().
mindspore.Tensor.random_	Fill the tensor with numbers sampled from a discrete uniform distribution over an interval $[from\_, to - 1]$.
mindspore.Tensor.random_categorical	For details, please refer to mindspore.ops.random_categorical().
mindspore.Tensor.ravel	Return a contiguous flattened tensor.
mindspore.Tensor.real	For details, please refer to mindspore.ops.real().
mindspore.Tensor.reciprocal	For details, please refer to mindspore.ops.reciprocal().
mindspore.Tensor.register_hook	Registers a backward hook for tensor.
mindspore.Tensor.remainder	Computes the remainder of *self* divided by *other* element-wise.

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.renorm	For details, please refer to mindspore.ops.renorm().
mindspore.Tensor.repeat	Copy the elements in each dimension of a Tensor based on the specified number of repetition times.
mindspore.Tensor.repeat_interleave	Repeat elements of a tensor along a dim, like mindspore.numpy.repeat().
mindspore.Tensor.reshape	For details, please refer to mindspore.ops.reshape().
mindspore.Tensor.reshape_as	Change the shape of the Tensor to the shape of *other* without changing the data.
mindspore.Tensor.resize	Changes shape and size of tensor in-place.
mindspore.Tensor.reverse	For details, please refer to mindspore.ops.flip().
mindspore.Tensor.reverse_sequence	For details, please refer to mindspore.ops.reverse_sequence().
mindspore.Tensor.roll	For details, please refer to mindspore.ops.roll().
mindspore.Tensor.rot90	For details, please refer to mindspore.ops.rot90().
mindspore.Tensor.round	For details, please refer to mindspore.ops.round().
mindspore.Tensor.rsqrt	For details, please refer to mindspore.ops.rsqrt().
mindspore.Tensor.scatter	Update the value in *src* to *self* according to the specified index.
mindspore.Tensor.scatter_	Update the value in *src* to update *self* according to the specified *index*.
mindspore.Tensor.scatter_add	Add all elements in *src* to the index specified by *index* to *self* along dimension specified by *dim*.
mindspore.Tensor.scatter_add_	Add all elements in *src* to the index specified by *index* to *self* along dimension specified by *dim*, *scatter_add* is an in-place operation.
mindspore.Tensor.scatter_div	For details, please refer to mindspore.ops.scatter_div().
mindspore.Tensor.scatter_max	For details, please refer to mindspore.ops.scatter_max().
mindspore.Tensor.scatter_min	For details, please refer to mindspore.ops.scatter_min().
mindspore.Tensor.scatter_mul	For details, please refer to mindspore.ops.scatter_mul().
mindspore.Tensor.scatter_sub	For details, please refer to mindspore.ops.tensor_scatter_sub().
mindspore.Tensor.searchsorted	For details, please refer to mindspore.ops.searchsorted().
mindspore.Tensor.select	Slices the *self* tensor along the selected dimension at the given index.
mindspore.Tensor.select_scatter	For details, please refer to mindspore.ops.select_scatter().
mindspore.Tensor.set_const_arg	Specify whether the tensor is a constant when it is used for the argument of a network.
mindspore.Tensor.sgn	For details, please refer to mindspore.ops.sgn().
mindspore.Tensor.shape	For details, please refer to mindspore.ops.shape().
mindspore.Tensor.short	Return a copy of the tensor, cast to int16 type, equivalent to self.astype(mstype.int16).
mindspore.Tensor.sigmoid	For details, please refer to mindspore.ops.sigmoid().
mindspore.Tensor.sign	For details, please refer to mindspore.ops.sign().
mindspore.Tensor.signbit	For details, please refer to mindspore.ops.signbit().
mindspore.Tensor.sin	For details, please refer to mindspore.ops.sin().
mindspore.Tensor.sinc	For details, please refer to mindspore.ops.sinc().
mindspore.Tensor.sinh	For details, please refer to mindspore.ops.sinh().

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.size	For details, please refer to mindspore.ops.size().
mindspore.Tensor.slice_scatter	For details, please refer to mindspore.ops.slice_scatter().
mindspore.Tensor.slogdet	For details, please refer to mindspore.ops.slogdet().
mindspore.Tensor.softmax	For details, please refer to mindspore.ops.softmax().
mindspore.Tensor.sort	Sorts the elements of the self tensor along the given dimension in the specified order.
mindspore.Tensor.split	Splits the Tensor into chunks along the given dim.
mindspore.Tensor.sqrt	For details, please refer to mindspore.ops.sqrt().
mindspore.Tensor.square	For details, please refer to mindspore.ops.square().
mindspore.Tensor.squeeze	For details, please refer to mindspore.ops.squeeze().
mindspore.Tensor.std	For details, please refer to mindspore.ops.std().
mindspore.Tensor.storage_offset	Tensor's offset in the underlying storage in terms of the number of storage elements.
mindspore.Tensor.stride	The stride to jump from one element to the next in the input dim.
mindspore.Tensor.strides	Return the tuple of bytes to step in each dimension when traversing a tensor.
mindspore.Tensor.sub	Subtracts scaled other value from self Tensor.
mindspore.Tensor.sub_	For details, please refer to mindspore.mint.sub().
mindspore.Tensor.subtract	This interface is deprecated from version 2.4 and will be removed in a future version.
mindspore.Tensor.sum	Calculate sum of Tensor elements over a given dim.
mindspore.Tensor.sum_to_size	Sum self Tensor to the *size*.
mindspore.Tensor.svd	For details, please refer to mindspore.ops.svd().
mindspore.Tensor.swapaxes	For details, please refer to mindspore.ops.swapaxes().
mindspore.Tensor.swapdims	For details, please refer to mindspore.ops.swapdims().
mindspore.Tensor.T	Return the transposed tensor.
mindspore.Tensor.t	Transpose *self* .
mindspore.Tensor.take	Takes elements from a tensor along an axis.
mindspore.Tensor.tan	For details, please refer to mindspore.ops.tan().
mindspore.Tensor.tanh	For details, please refer to mindspore.ops.tanh().
mindspore.Tensor.tensor_split	For details, please refer to mindspore.ops.tensor_split().
mindspore.Tensor.tile	Replicates an tensor with given dims times.
mindspore.Tensor.to	Performs tensor dtype conversion.
mindspore.Tensor.to_coo	Convert a Tensor to COOTensor.
mindspore.Tensor.to_csr	Convert a Tensor to CSRTensor.
mindspore.Tensor.tolist	Convert a Tensor to List.
mindspore.Tensor.topk	Finds the *k* largest or smallest element along the given dimension and returns its value and corresponding index.
mindspore.Tensor.trace	Return the sum along diagonals of the tensor.
mindspore.Tensor.transpose	Interchange two axes of a tensor.
mindspore.Tensor.triangular_solve	For details, please refer to mindspore.mint.triangular_solve().
mindspore.Tensor.tril	For details, please refer to mindspore.ops.tril().
mindspore.Tensor.triu	For details, please refer to mindspore.ops.triu().
mindspore.Tensor.true_divide	Alias for Tensor.div() with $rounding\_mode = None$.
mindspore.Tensor.trunc	For details, please refer to mindspore.ops.trunc().
mindspore.Tensor.type	Change the dtype of the Tensor to the *dtype* .
mindspore.Tensor.type_as	Returns self tensor cast to the type of the with the input other tensor.

continues on next page

Table 2 – continued from previous page

mindspore.Tensor.unbind	For details, please refer to mindspore.ops.unbind().
mindspore.Tensor.unfold	For details, please refer to mindspore.ops.unfold().
mindspore.Tensor.uniform	Generates random numbers that follows a uniform distribution within the half-open interval $[from\_, to)$.
mindspore.Tensor.uniform_	Update the *self* Tensor in place by generating random numbers sampled from uniform distribution in the half-open interval $[from\_, to)$.
mindspore.Tensor.unique	Returns the unique elements of *self*.
mindspore.Tensor.unique_consecutive	For details, please refer to mindspore.ops.unique_consecutive().
mindspore.Tensor.unsorted_segment_max	For details, please refer to mindspore.ops.unsorted_segment_max().
mindspore.Tensor.unsorted_segment_min	For details, please refer to mindspore.ops.unsorted_segment_min().
mindspore.Tensor.unsorted_segment_prod	For details, please refer to mindspore.ops.unsorted_segment_prod().
mindspore.Tensor.unsqueeze	For details, please refer to mindspore.ops.unsqueeze().
mindspore.Tensor.var	Compute the variance along the specified axis.
mindspore.Tensor.view	Reshape the tensor according to the input shape.
mindspore.Tensor.view_as	View *self* Tensor as the same shape as *other* .
mindspore.Tensor.vsplit	For details, please refer to mindspore.ops.vsplit().
mindspore.Tensor.where	For details, please refer to mindspore.ops.where().
mindspore.Tensor.xdivy	For details, please refer to mindspore.ops.xdivy().
mindspore.Tensor.xlogy	For details, please refer to mindspore.ops.xlogy().
mindspore.Tensor.zero_	Return a tensor filled with zeros.

mindspore.Tensor.abs

Tensor.**abs**() → *Tensor*

For details, please refer to mindspore.ops.abs().

mindspore.Tensor.absolute

Tensor.**absolute**() → *Tensor*

Alias for Tensor.abs().

mindspore.Tensor.acos

Tensor.**acos**() → *Tensor*

For details, please refer to mindspore.ops.acos().

mindspore.Tensor.acosh

Tensor.**acosh**() → *Tensor*

For details, please refer to *mindspore.ops.acosh()*.

mindspore.Tensor.add_

Tensor.**add_**(*other*) → *Tensor*

In-place version of *mindspore.Tensor.add()*.

mindspore.Tensor.add

Tensor.**add**(*other*) → *Tensor*

Adds other value to *self* element-wise.

$$out_i = self_i + other_i$$

Note:

- When *self* and *other* have different shapes, they must be able to broadcast to a common shape.
- *self* and *other* can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- *self* and *other* comply with the implicit type conversion rules to make the data types consistent.
- The dimension of *self* should be greater than or equal to 1.

Parameters

other (*Union[Tensor, number.Number, bool]*) – *other* is a number.Number or a bool or a tensor whose data type is number or bool_.

Returns

Tensor with a shape that is the same as the broadcasted shape of *self* and *other*, and the data type is the one with higher precision or higher digits between *self* and *other*.

Raises

TypeError – If *other* is not one of the following: Tensor, number.Number, bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> # case 1: x and y are both Tensor.
>>> x = Tensor(np.array([1, 2, 3]).astype(np.float32))
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> output = Tensor.add(x, y)  # x.add(y)
>>> print(output)
[5. 7. 9.]
>>> # case 2: x is a scalar and y is a Tensor
>>> x = Tensor(1, mindspore.int32)
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> output = Tensor.add(x, y)  # x.add(y)
>>> print(output)
[5. 6. 7.]
>>> # the data type of x is int32, the data type of y is float32,
>>> # and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

`Tensor.`**`add`**(*other*, *, *alpha=1*) → *Tensor*

Adds scaled other value to *self*.

$$out_i = self_i + alpha \times other_i$$

Note:

- When *self* and *other* have different shapes, they must be able to broadcast to a common shape.
- *self*, *other* and alpha comply with the implicit type conversion rules to make the data types consistent.

Parameters

`other` (*`Union[Tensor, number.Number, bool]`*) – *other* is a number.Number or a bool or a tensor whose data type is number or bool_.

Keyword Arguments

`alpha` (*`number.Number`*) – A scaling factor applied to *other*, default 1.

Returns

Tensor with a shape that is the same as the broadcasted shape of the *self* and *other*, and the data type is the one with higher precision or higher digits among *self*, *other* and *alpha*.

Raises

- **`TypeError`** – If the type of *other* or *alpha* is not one of the following: Tensor, number.Number, bool.
- **`TypeError`** – If *alpha* is of type float but *self* and *other* are not of type float.
- **`TypeError`** – If *alpha* is of type bool but *self* and *other* are not of type bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(1, mindspore.int32)
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> alpha = 0.5
>>> output = Tensor.add(x, y, alpha=alpha)  # x.add(y, alpha=alpha)
>>> print(output)
[3. 3.5 4.]
>>> # the data type of x is int32, the data type of y is float32,
>>> # alpha is a float, and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

mindspore.Tensor.addbmm

Tensor.**addbmm**(*batch1*, *batch2*, *, *beta=1*, *alpha=1*) → *Tensor*

For details, please refer to *mindspore.ops.addbmm()*.

mindspore.Tensor.addcdiv

Tensor.**addcdiv**(*tensor1*, *tensor2*, *, *value=1*) → *Tensor*

For details, please refer to *mindspore.ops.addcdiv()*.

Supported Platforms:

Ascend

mindspore.Tensor.addcmul

Tensor.**addcmul**(*tensor1*, *tensor2*, *value=1*)

For details, please refer to *mindspore.ops.addcmul()*.

mindspore.Tensor.addmm

Tensor.**addmm**(*mat1*, *mat2*, *, *beta=1*, *alpha=1*) → *Tensor*

For details, please refer to *mindspore.ops.addmm()*.

mindspore.Tensor.addmm_

Tensor.**addmm_**(*mat1*, *mat2*, *, *beta=1*, *alpha=1*)

In-place version of *mindspore.Tensor.addmm()*.

Warning: This is an experimental API that is subject to change or deletion.

mindspore.Tensor.addmv

Tensor.**addmv**(*mat*, *vec*, *, *beta=1*, *alpha=1*) → *Tensor*

For details, please refer to *mindspore.ops.addmv()*.

Supported Platforms:

Ascend

mindspore.Tensor.addr

Tensor.**addr**(*vec1*, *vec2*, *beta=1*, *alpha=1*)

For details, please refer to *mindspore.ops.addr()*.

mindspore.Tensor.adjoint

Tensor.**adjoint**()

For details, please refer to *mindspore.ops.adjoint()*.

mindspore.Tensor.all

Tensor.**all**(*axis=None*, *keep_dims=False*) → *Tensor*

Tests if all element in tensor evaluates to *True* along the given axes.

Parameters

- **axis** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. If None , all dimensions are reduced. Default None .
- **keep_dims** (*bool, optional*) –Whether the output tensor has dim retained or not. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.Tensor.all tests along all the axes.
>>> x.all()
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 2: Reduces a dimension along axis 1, with keep_dims False.
>>> x.all(axis=1)
Tensor(shape=[2], dtype=Bool, value= [False,  True])
>>>
>>> # case 3: Reduces a dimension along axis (0, 1), with keep_dims False.
>>> x.all(axis=(0,1))
```

(continues on next page)

(continued from previous page)

```
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 4: Reduces a dimension along axis [0, 1], with keep_dims True.
>>> x.all(axis=[0,1], keep_dims=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[False]])
```

`Tensor.all`(*dim=None*, *keepdim=False*) → *Tensor*

Tests if all element in tensor evaluates to *True* along the given axes.

Parameters

- **dim** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. If `None` , all dimensions are reduced. Default `None` .
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained or not. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.Tensor.all tests along all the axes.
>>> x.all()
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 2: Reduces a dimension along dim 1, with keepdim False.
>>> x.all(dim=1)
Tensor(shape=[2], dtype=Bool, value= [False,  True])
>>>
>>> # case 3: Reduces a dimension along dim (0, 1), with keepdim False.
>>> x.all(dim=(0,1))
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 4: Reduces a dimension along dim [0, 1], with keepdim True.
>>> x.all(dim=[0,1], keepdim=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[False]])
```

mindspore.Tensor.allclose

`Tensor.allclose`(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*) → *Tensor*

Returns a new Tensor with boolean elements representing if each element of *self* is "close" to the corresponding element of *other*. Closeness is defined as:

$$|self - other| <= atol + rtolx|other|$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **other** (Tensor) – Tensor to compare. Dtype must be same as *self*.
- **rtol** (*Union[float, int, bool], optional*) – Relative tolerance. Default: `1e-05` .
- **atol** (*Union[float, int, bool], optional*) – Absolute tolerance. Default: `1e-08` .
- **equal_nan** (*bool, optional*) – If `True` , then two NaNs will be considered equal. Default: `False`.

Returns

A bool Scalar.

Raises

- **TypeError** – *self* or *other* is not Tensor.
- **TypeError** – Data types of *self* and *other* are not in the list of supported types.
- **TypeError** – *atol* or *rtol* is not float, int or bool.
- **TypeError** – *equal_nan* is not bool.
- **TypeError** – *self* and *other* have different dtypes.
- **ValueError** – *self* and *other* cannot broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([1.3, 2.1, 3.2, 4.1, 5.1]), mindspore.float16)
>>> other = Tensor(np.array([1.3, 3.3, 2.3, 3.1, 5.1]), mindspore.float16)
>>> output = input.allclose(other)
>>> print(output)
False
```

mindspore.Tensor.amax

Tensor.**amax**(*axis=None*, *keepdims=False*, *, *initial=None*, *where=None*)
For details, please refer to *mindspore.ops.amax()*.

mindspore.Tensor.amin

Tensor.**amin**(*axis=None*, *keepdims=False*, *, *initial=None*, *where=None*)
For details, please refer to *mindspore.ops.amin()*.

mindspore.Tensor.aminmax

Tensor.**aminmax**(*, *axis=0*, *keepdims=False*)
For details, please refer to *mindspore.ops.aminmax()*.

mindspore.Tensor.any

Tensor.**any**(*axis=None*, *keep_dims=False*) → *Tensor*
Tests if any element in tensor evaluates to *True* along the given axes.

Parameters

- **axis** (*Union[int, tuple(int), list(int), Tensor], optional*) – The dimensions to reduce. If `None` , all dimensions are reduced. Default `None` .
- **keep_dims** (*bool, optional*) – Whether the output tensor has dim retained or not. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.Tensor.any tests along all the axes.
>>> x.any()
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 2: Reduces a dimension along axis 1, with keep_dims False.
>>> x.any(axis=1)
Tensor(shape=[2], dtype=Bool, value= [ True,  True])
>>>
>>> # case 3: Reduces a dimension along axis (0, 1), with keep_dims False.
>>> x.any(axis=(0,1))
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 4: Reduces a dimension along axis [0, 1], with keep_dims True.
```

(continues on next page)

(continued from previous page)

```
>>> x.any(axis=[0,1], keep_dims=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[ True]])
```

`Tensor.`**`any`**(*dim=None*, *keepdim=False*) → *Tensor*

Tests if any element in tensor evaluates to *True* along the given axes.

Parameters

- **dim** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. If `None` , all dimensions are reduced. Default `None` .
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained or not. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.Tensor.any tests along all the axes.
>>> x.any()
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 2: Reduces a dimension along dim 1, with keepdim False.
>>> x.any(dim=1)
Tensor(shape=[2], dtype=Bool, value= [ True,  True])
>>>
>>> # case 3: Reduces a dimension along dim (0, 1), with keepdim False.
>>> x.any(dim=(0,1))
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 4: Reduces a dimension along dim [0, 1], with keepdim True.
>>> x.any(dim=[0,1], keepdim=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[ True]])
```

mindspore.Tensor.angle

`Tensor.`**`angle`**()

For details, please refer to *mindspore.ops.angle()*.

mindspore.Tensor.approximate_equal

Tensor.**approximate_equal**(*other*, *tolerance=1e-5*)
For details, please refer to *mindspore.ops.approximate_equal()*, The parameter *other* of current interface is the same as the parameter *y* of the reference interface.

mindspore.Tensor.arccos

Tensor.**arccos**() → *Tensor*
Alias for *mindspore.Tensor.acos()*.

mindspore.Tensor.arccosh

Tensor.**arccosh**() → *Tensor*
Alias for *mindspore.Tensor.acosh()*.

mindspore.Tensor.arcsin

Tensor.**arcsin**() → *Tensor*
Alias for *mindspore.Tensor.asin()*.

mindspore.Tensor.arcsinh

Tensor.**arcsinh**() → *Tensor*
Alias for *mindspore.Tensor.asinh()*.

mindspore.Tensor.arctan

Tensor.**arctan**() → *Tensor*
Alias for *mindspore.Tensor.atan()*.

mindspore.Tensor.arctan2

Tensor.**arctan2**(*other*) → *Tensor*
Alias for *Tensor.atan2()*.

mindspore.Tensor.arctanh

Tensor.**arctanh**() → *Tensor*
Alias for *mindspore.Tensor.atanh()*.

mindspore.Tensor.argmax

Tensor.**argmax**(*axis=None*, *keepdims=False*) → *Tensor*

Return the indices of the maximum values along the given axis of the tensor.

Parameters

- **axis** (*Union[int, None], optional*) –Specify the axis for computation. If None , compute all elements in the tensor. Default None .
- **keepdims** (*bool, optional*) –Whether the output tensor has dim retained. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[9, 3, 4, 5],
...                       [5, 2, 7, 4],
...                       [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum of all elements.
>>> x.argmax()
Tensor(shape=[], dtype=Int64, value= 0)
>>>
>>> # case 2: Compute the maximum along axis 1.
>>> x.argmax(axis=1)
Tensor(shape=[3], dtype=Int64, value= [0, 2, 0])
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> x.argmax(axis=1, keepdims=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[0],
 [2],
 [0]])
```

Tensor.**argmax**(*dim=None*, *keepdim=False*) → *Tensor*

Return the maximum values along the given dimension of the tensor.

Parameters

- **dim** (*Union[int, None], optional*) –Specify the dim for computation. If None , compute all elements in the tensor.
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[9, 3, 4, 5],
...                       [5, 2, 7, 4],
...                       [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum of all elements.
>>> x.argmax()
Tensor(shape=[], dtype=Int64, value= 0)
>>>
>>> # case 2: Compute the maximum along dim 1.
>>> x.argmax(dim=1)
Tensor(shape=[3], dtype=Int64, value= [0, 2, 0])
>>>
>>> # case 3: If keepdim=True, the output shape will be same of that of the input.
>>> x.argmax(dim=1, keepdim=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[0],
 [2],
 [0]])
```

mindspore.Tensor.argmax_with_value

Tensor.**argmax_with_value**(*axis=0*, *keep_dims=False*)

Return the maximum values and their indices along the given axis of the tensor.

Parameters

- **axis** (*Union[int, None], optional*) – Specify the axis for computation. If `None` , compute all elements in the tensor. Default `0` .
- **keep_dims** (*bool, optional*) – Whether the output tensor has dim retained. Default `False` .

Returns

Tuple(max, max_indices) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[9, 3, 4, 5],
...                       [5, 2, 7, 4],
...                       [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum along axis 0.
>>> x.argmax_with_value()
(Tensor(shape=[4], dtype=Int64, value= [9, 3, 7, 6]),
 Tensor(shape=[4], dtype=Int64, value= [0, 0, 1, 2]))
>>>
>>> # case 2: Compute the maximum along axis 1.
>>> x.argmax_with_value(axis=1)
(Tensor(shape=[3], dtype=Int64, value= [9, 7, 8]),
 Tensor(shape=[3], dtype=Int64, value= [0, 2, 0]))
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # case 3: If keep_dims=True, the output shape will be same of that of the input.
>>> x.argmax_with_value(axis=1, keep_dims=True)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[9],
  [7],
  [8]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[0],
  [2],
  [0]]))
>>>
>>> # case 4: If axis=None, compute the maximum of all elements.
>>> x.argmax_with_value(axis=None, keep_dims=True)
(Tensor(shape=[], dtype=Int64, value= 9),
 Tensor(shape=[], dtype=Int64, value= 0))
```

mindspore.Tensor.argmin

`Tensor.`**`argmin`**(*axis=None*, *keepdims=False*) → *Tensor*

Returns the indices of the minimum values along the given axis of the tensor.

Parameters

- **`axis`** (*`Union[int, None], optional`*) – Specify the axis for computation. If `None` , compute all elements in the tensor. Default `None` .
- **`keepdims`** (*`bool, optional`*) – Whether the output tensor has dim retained. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[2, 5, 1, 6],
...                       [3, -7, -2, 4],
...                       [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum of all elements.
>>> x.argmin()
Tensor(shape=[], dtype=Int32, value= 5)
>>>
>>> # case 2: Compute the minimum along axis 1.
>>> x.argmin(axis=1)
Tensor(shape=[3], dtype=Int32, value= [2, 1, 1])
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> x.argmin(axis=1, keepdims=True)
Tensor(shape=[3, 1], dtype=Int32, value=
[[2],
```

(continues on next page)

(continued from previous page)

```
 [1],
 [1]])
```

`Tensor.`**`argmin`**(*dim=None*, *keepdim=False*) → *Tensor*

Returns the indices of the minimum values along the given axis of the tensor.

Parameters

- **dim** (*Union[int, None], optional*) –Specify the axis for computation. If `None` , compute all elements in the tensor.
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[2, 5, 1, 6],
...                       [3, -7, -2, 4],
...                       [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum of all elements.
>>> x.argmin()
Tensor(shape=[], dtype=Int32, value= 5)
>>>
>>> # case 2: Compute the minimum along dim 1.
>>> x.argmin(dim=1)
Tensor(shape=[3], dtype=Int32, value= [2, 1, 1])
>>>
>>> # case 3: If keepdim=True, the output shape will be same of that of the input.
>>> x.argmin(dim=1, keepdim=True)
Tensor(shape=[3, 1], dtype=Int32, value=
[[2],
 [1],
 [1]])
```

mindspore.Tensor.argmin_with_value

`Tensor.`**`argmin_with_value`**(*axis=0*, *keep_dims=False*)

Return the minimum values and their indices along the given axis of the tensor.

Parameters

- **axis** (*Union[int, None], optional*) –Specify the axis for computation. If `None` , compute all elements in the tensor. Default `0` .
- **keep_dims** (*bool, optional*) –Whether the output tensor has dim retained. Default `False` .

Returns

Tuple(min, min_indices) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[2, 5, 1, 6],
...                       [3, -7, -2, 4],
...                       [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum along axis 0.
>>> x.argmin_with_value()
(Tensor(shape=[4], dtype=Int64, value= [ 2, -7, -2, -3]),
 Tensor(shape=[4], dtype=Int64, value= [0, 1, 1, 2]))
>>>
>>> # case 2: Compute the minimum along axis 1.
>>> x.argmin_with_value(axis=1)
(Tensor(shape=[3], dtype=Int64, value= [ 1, -7, -4]),
 Tensor(shape=[3], dtype=Int64, value= [2, 1, 1]))
>>>
>>> # case 3: If keep_dims=True, the output shape will be same of that of the input.
>>> x.argmin_with_value(axis=1, keep_dims=True)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[ 1],
  [-7],
  [-4]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[2],
  [1],
  [1]]))
>>>
>>> # case 4: If axis=None, compute the minimum of all elements.
>>> x.argmin_with_value(axis=None, keep_dims=True)
(Tensor(shape=[], dtype=Int64, value= -7),
 Tensor(shape=[], dtype=Int64, value= 0))
```

mindspore.Tensor.argsort

`Tensor.`**`argsort`**(*axis=- 1*, *descending=False*) → *Tensor*

Sorts *self* along the given dimension in specified order and return the sorted indices.

Parameters

- **`axis`** (*int, optional*) – The axis to sort along. Default: `-1` , means the last dimension. The Ascend backend only supports sorting the last dimension.
- **`descending`** (*bool, optional*) – The sort order. If *descending* is True then the elements are sorted in descending order by value. Otherwise sort in ascending order. Default: `False` .

Returns

Tensor, the indices of sorted *self*. Data type is int32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> sort = Tensor.argsort(x)  # x.argsort()
>>> print(sort)
[[2 1 0]
 [2 0 1]
 [0 1 2]]
```

Tensor.**argsort**(*dim=- 1*, *descending=False*, *stable=False*) → *Tensor*

Sorts *self* along the given dimension in specified order and return the sorted indices.

Warning: This is an experimental optimizer API that is subject to deletion or change.

Parameters

- **dim** (*int, optional*) –The dim to sort along. Default: `-1` , means the last dimension. The Ascend backend only supports sorting the last dimension.
- **descending** (*bool, optional*) –The sort order. If *descending* is `True` then the elements are sorted in descending order by value. Otherwise sort in ascending order. Default: `False` .
- **stable** (*bool, optional*) –Whether to use stable sorting algorithm. Default: `False`.

Returns

Tensor, the indices of sorted *self*. Data type is int64.

Raises

- **ValueError** –If *dim* is out of range.
- **TypeError** –If dtype of *dim* is not int32.
- **TypeError** –If dtype of *descending* is not bool.
- **TypeError** –If dtype of *stable* is not bool.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> sort = Tensor.argsort(x)  # x.argsort()
>>> print(sort)
[[2 1 0]
 [2 0 1]
 [0 1 2]]
```

mindspore.Tensor.argwhere

Tensor.**argwhere**()
For details, please refer to *mindspore.ops.argwhere()*.

mindspore.Tensor.asin

Tensor.**asin**() → *Tensor*
For details, please refer to *mindspore.ops.asin()*.

mindspore.Tensor.asinh

Tensor.**asinh**() → *Tensor*
For details, please refer to *mindspore.ops.asinh()*.

mindspore.Tensor.asnumpy

Tensor.**asnumpy**()
Convert tensor to numpy array. Returns self tensor as a NumPy ndarray. This tensor and the returned ndarray share the same underlying storage. Changes to self tensor will be reflected in the ndarray.

Returns

A numpy ndarray which shares the same underlying storage with the tensor.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([1, 2], dtype=np.float32))
>>> y = x.asnumpy()
>>> y[0] = 11
>>> print(x)
[11.  2.]
>>> print(y)
[11.  2.]
```

mindspore.Tensor.assign_value

Tensor.**assign_value**(*value*)
Assign another tensor value to this tensor.

Parameters

value (Tensor) – Tensor for assignment.

Returns

Tensor, Tensor that's been assigned.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor([1, 2, 3, 4])
>>> y = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.assign_value(y)
>>> print(x)
[[1 2]
[3 4]]
```

mindspore.Tensor.astype

Tensor.**astype**(*dtype*, *copy=True*)

Return a copy of the tensor, cast to a specified type.

Parameters

- **dtype** (Union[*mindspore.dtype*, numpy.dtype, str]) –Designated tensor dtype, can be in format of *mindspore.dtype.float32* or *numpy.float32* or *float32*.
- **copy** (*bool, optional*) –By default, astype always returns a newly allocated tensor. If this is set to `false` , the input tensor is returned instead of a copy. Default: `True` .

Returns

Tensor, with the designated dtype.

Raises

TypeError –If the specified dtype cannot be understood.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.ones((1,2,2,1), dtype=np.float32))
>>> x = x.astype("int32")
>>> print(x.dtype)
Int32
```

mindspore.Tensor.atan

Tensor.**atan**() → *Tensor*

For details, please refer to *mindspore.ops.atan()*.

mindspore.Tensor.atan2

Tensor.**atan2**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.atan2()*.

mindspore.Tensor.atanh

Tensor.**atanh**() → *Tensor*

For details, please refer to *mindspore.ops.atanh()*.

mindspore.Tensor.baddbmm

Tensor.**baddbmm**(*batch1*, *batch2*, *, *beta=1*, *alpha=1*) → *Tensor*

For details, please refer to *mindspore.ops.baddbmm()*.

mindspore.Tensor.bernoulli

Tensor.**bernoulli**(*, *generator=None*)

For details, please refer to *mindspore.mint.bernoulli()*.

mindspore.Tensor.bfloat16

Tensor.**bfloat16**()

Converts input tensor dtype to *bfloat16*.

Returns

Tensor, converted to the *bfloat16* dtype.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.int32)
>>> output = input_x.bfloat16()
>>> print(output.dtype)
BFloat16
```

mindspore.Tensor.bincount

Tensor.**bincount**(*weights=None*, *minlength=0*) → *Tensor*

For details, please refer to *mindspore.ops.bincount()*.

mindspore.Tensor.bitwise_and

Tensor.**bitwise_and**(*other*) → *Tensor*

Returns bitwise *and* of two tensors element-wise.

Note: *self* and *other* comply with the type conversion rules to make the data types consistent.

Parameters

other (Tensor, *Number.number*) – The shape is the same as the *self* or can be broadcast to the shape of *self*.

Returns

Tensor, has the same type as the *self* and has the same shape as after broadcasting.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = input.bitwise_and(other)
>>> print(output)
[ 0  0  1 -1  1  0  1]
```

mindspore.Tensor.bitwise_not

Tensor.**bitwise_not**() → *Tensor*

Returns bitwise *not* of *self*.

Warning: This is an experimental API that is subject to change or deletion.

Returns

Tensor, has the same shape and type as *self*.

Raises

- **TypeError** – If *self* is not a Tensor.
- **RuntimeError** – If dtype of *self* is not int or bool.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([True, False, True, False]))
>>> output = input.bitwise_not()
>>> print(output)
[False True False True]
```

mindspore.Tensor.bitwise_left_shift

Tensor.**bitwise_left_shift**(*other*)

For details, please refer to *mindspore.ops.bitwise_left_shift()*.

mindspore.Tensor.bitwise_or

Tensor.**bitwise_or**(*other*) → *Tensor*

Returns bitwise *or* of two tensors element-wise.

Note: *self* and *other* comply with the type conversion rules to make the data types consistent.

Parameters

other (Tensor, *Number.number*) – The shape is the same as the *self* or can be broadcast to the shape of *self*.

Returns

Tensor, has the same type as the *self* and has the same shape as after broadcasting.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = input.bitwise_or(other)
>>> print(output)
[ 0  1  1 -1 -1  3  3]
```

mindspore.Tensor.bitwise_right_shift

Tensor.**bitwise_right_shift**(*other*)
For details, please refer to *mindspore.ops.bitwise_right_shift()*.

mindspore.Tensor.bitwise_xor

Tensor.**bitwise_xor**(*other*) → *Tensor*
Returns bitwise *xor* of two tensors element-wise.

Note: *self* and *other* comply with the type conversion rules to make the data types consistent.

Parameters

other (Tensor, *Number.number*) – The shape is the same as the *self* or can be broadcast to the shape of *self*.

Returns

Tensor, has the same type as the *self* and has the same shape as after broadcasting.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = input.bitwise_xor(other)
>>> print(output)
[ 0  1  0  0 -2  3  2]
```

mindspore.Tensor.bmm

Tensor.**bmm**(*mat2*)
For details, please refer to *mindspore.ops.bmm()*.

mindspore.Tensor.bool

Tensor.**bool**()
Converts input tensor dtype to *bool*. If the value in tensor is zero, it will be *False*, otherwise it will be *True*.

Returns

Tensor, converted to the *bool* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.float32)
>>> output = input_x.bool()
>>> print(output.dtype)
Bool
```

mindspore.Tensor.broadcast_to

Tensor.**broadcast_to**(*shape*)

For details, please refer to *mindspore.ops.broadcast_to()*.

mindspore.Tensor.byte

Tensor.**byte**()

Converts input tensor dtype to *uint8*.

Returns

Tensor, converted to the *uint8* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.float32)
>>> output = input_x.byte()
>>> print(output.dtype)
UInt8
```

mindspore.Tensor.cauchy

Tensor.**cauchy**(*median=0.0*, *sigma=1.0*)

Fills the tensor with numbers drawn from the Cauchy distribution. It is defined as follows:

$$f(x) = \frac{1}{\pi} \frac{\sigma}{(x - median)^2 + \sigma^2}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **median** (*float, optional*) – the location parameter, specifying the location of the peak of the distribution. Default: 0.0.
- **sigma** (*float, optional*) – the scale parameter which specifies the half-width at half-maximum. Default: 1.0.

Returns

Tensor. A Tensor with the same type and shape of input.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> x = mindspore.Tensor(np.zeros((1, 2)), dtype=mindspore.float32)
>>> x.cauchy()
Tensor(shape=[1, 2], dtype=Float32, value=
[[8.79836142e-01, 9.37541723e-01]])
```

mindspore.Tensor.ceil

`Tensor.ceil()` → *Tensor*

For details, please refer to *mindspore.ops.ceil()*.

mindspore.Tensor.cholesky

`Tensor.cholesky`(*upper=False*)

For details, please refer to *mindspore.ops.cholesky()*.

mindspore.Tensor.cholesky_solve

`Tensor.cholesky_solve`(*input2*, *upper=False*)

For details, please refer to *mindspore.ops.cholesky_solve()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.choose

`Tensor.choose`(*choices*, *mode='clip'*)

Construct a tensor from an index tensor and a list of tensors to choose from.

Parameters

- **choices** (*Union[tuple, list, Tensor]*) – Choice tensors. The input tensor and all of the *choices* must be broadcasted to the same shape. If *choices* is itself a tensor, then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the "sequence".

- **mode** (*str, optional*) –Specifies how indices outside `[0, n-1]` will be treated. Support `'raise'`, `'wrap'`, `'clip'`.
  - `raise`: Raises an error;
  - `wrap`: Wraps around;
  - `clip`: Clips to the range. The values greater than n-1 will be mapped to n-1. Note that this mode disables indexing with negative numbers.

  Default: `'clip'`.

Returns

Tensor, the merged result.

Raises

ValueError –If the input tensor and any of the *choices* cannot be broadcast.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23], [30, 31, 32, 33]]
>>> x = Tensor(np.array([2, 3, 1, 0]))
>>> print(x.choose(choices))
[20 31 12  3]
```

mindspore.Tensor.chunk

`Tensor.`**chunk**(*chunks*, *dim=0*) → Tuple of Tensors

Cut the self Tensor into *chunks* sub-tensors along the specified dimension.

Note: The number of sub-tensors returned by this function may be less than the number of sub-tensors specified by *chunks*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **chunks** (*int*) –Number of sub-tensors to cut.
- **dim** (*int, optional*) –Specify the dimensions that you want to split. Default: `0` .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –The sum of *chunks* is not int.
- **TypeError** –If argument *dim* is not int.

- **ValueError** –If argument *dim* is out of range of $[-self.ndim, self.ndim)$.
- **ValueError** –If argument *chunks* is not positive number.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.arange(9).astype("float32"))
>>> output = input_x.chunk(3, dim=0)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
    Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
    Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

`Tensor.`**`chunk`**(*chunks*, *axis=0*) → Tuple of Tensors

Cut the self Tensor into *chunks* sub-tensors along the specified axis.

Note: This function may return less than the specified number of chunks!

Parameters

- **chunks** (*int*) –Number of sub-tensors to cut.
- **axis** (*int, optional*) –Specify the dimensions that you want to split. Default: `0` .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –The sum of *chunks* is not int.
- **TypeError** –If argument *axis* is not int.
- **ValueError** –If argument *axis* is out of range of $[-self.ndim, self.ndim)$.
- **ValueError** –If argument *chunks* is not positive number.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.arange(9).astype("float32"))
>>> output = input_x.chunk(3, axis=0)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
    Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
    Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

mindspore.Tensor.clamp

Tensor.**clamp**(*min=None*, *max=None*) → *Tensor*

For details, please refer to *mindspore.ops.clamp()*.

mindspore.Tensor.clamp_

Tensor.**clamp_**(*min=None*, *max=None*)

In-place version of *mindspore.Tensor.clamp()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.clip

Tensor.**clip**(*min=None*, *max=None*) → *Tensor*

Alias for *mindspore.Tensor.clamp()*.

mindspore.Tensor.clone

Tensor.**clone**() → *Tensor*

Returns a copy of self.

> **Warning:** This is an experimental API that is subject to change or deletion.

Note: This function is differentiable, and gradients will flow back directly from the calculation result of the function to the *self*.

Returns

Tensor, with the same data, shape and type as *self*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.ones((3,3)).astype("float32"))
>>> output = input.clone()
>>> print(output)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

mindspore.Tensor.col2im

Tensor.**col2im**(*output_size*, *kernel_size*, *dilation*, *padding_value*, *stride*)

For details, please refer to *mindspore.ops.col2im()*.

mindspore.Tensor.conj

Tensor.**conj**()

For details, please refer to *mindspore.ops.conj()*.

mindspore.Tensor.contiguous

Tensor.**contiguous**()

Converts a Tensor into a continuous-memory Tensor that contains the same data as the original Tensor.

Returns

A contiguous in memory tensor containing the same data as self tensor.

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor([[1, 2, 3], [4, 5, 6]], dtype=ms.float32)
>>> y = ops.transpose(x, (1, 0))
>>> z = y.contiguous()
>>> print(z.is_contiguous())
True
```

mindspore.Tensor.copy

Tensor.**copy**()
Return a copy of the tensor.

Note: The current implementation does not support *order* argument.

Returns

Copied tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> a = Tensor(np.ones((3,3)).astype("float32"))
>>> output = a.copy()
>>> print(output)
[[1. 1. 1.]
[1. 1. 1.]
[1. 1. 1.]]
```

mindspore.Tensor.copy_

Tensor.**copy_**(*src*, *non_blocking=False*) → *Tensor*
Copies the elements from *src* into *self* tensor and returns *self*.

Warning: This is an experimental API that is subject to change or deletion. The *src* tensor must be broadcastable with the *self* tensor. It may be of a different data type.

Parameters

- **src** (Tensor) – the source tensor to copy from.
- **non_blocking** (*bool, optional*) – no effect currently. Default: False.

Returns

Return self Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> a = Tensor(np.ones((3, 3)).astype("float32"))
>>> b = Tensor(np.zeros((3, 3)).astype("float32"))
>>> a.copy_(b)
>>> print(a)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.Tensor.copysign

Tensor.**copysign**(*other*)

For details, please refer to *mindspore.ops.copysign()*.

mindspore.Tensor.cos

Tensor.**cos**() → *Tensor*

For details, please refer to *mindspore.ops.cos()*.

mindspore.Tensor.cosh

Tensor.**cosh**() → *Tensor*

For details, please refer to *mindspore.ops.cosh()*.

mindspore.Tensor.count_nonzero

Tensor.**count_nonzero**(*dim=None*) → *Tensor*

Counts the number of non-zero values in the tensor input along the given dim. If no dim is specified then all non-zeros in the tensor are counted.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

dim (*Union[None, int, tuple(int), list(int)], optional*) – The dimension to reduce. Default value: None, which indicates that the number of non-zero elements is calculated. If *dim* is None, all elements in the tensor are summed up.

Returns

Tensor, number of nonzero element across dim specified by *dim*.

Raises

- **TypeError** – If *dim* is not int, tuple(int), list(int) or None.
- **ValueError** – If any value in *dim* is not in range $[-self.ndim, self.ndim)$.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> import mindspore
>>> # case 1: each value specified.
>>> x = Tensor(np.array([[0, 1, 0], [1, 1, 0]]).astype(np.float32))
>>> nonzero_num = x.count_nonzero(dim=[0, 1])
>>> print(nonzero_num)
[[3]]
>>> # case 2: all value is default.
>>> nonzero_num = x.count_nonzero()
>>> print(nonzero_num)
3
>>> # case 3: dim value was specified 0.
>>> nonzero_num = x.count_nonzero(dim=[0,])
>>> print(nonzero_num)
[1 2 0]
>>> # case 4: dim value was specified 1.
>>> nonzero_num = x.count_nonzero(dim=[1,])
>>> print(nonzero_num)
[1 2]
```

`Tensor.`**`count_nonzero`**(*axis=()*, *keep_dims=False*, *dtype=None*) → *Tensor*

Count number of nonzero elements across axis of input tensor.

Parameters

- **`axis`** (*`Union[int, tuple(int), list(int)], optional`*) – The dimensions to reduce. Default: `()`, reduce all dimensions.
- **`keep_dims`** (*`bool, optional`*) – Whether to maintain dimensions specified by *axis*. If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default: `False`.
- **`dtype`** (*`Union[Number, mindspore.bool_], optional`*) – The data type of the output tensor. Default: `None`.

Returns

Tensor, number of nonzero element across axis specified by *axis*. The data type is specified by *dtype*.

Raises

- **`TypeError`** – If *axis* is not int, tuple or list.
- **`ValueError`** – If any value in *axis* is not in range $[-self.ndim, self.ndim)$.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> import mindspore
>>> # case 1: each value specified.
>>> x = Tensor(np.array([[0, 1, 0], [1, 1, 0]]).astype(np.float32))
>>> nonzero_num = x.count_nonzero(x=x, axis=[0, 1], keep_dims=True, dtype=mindspore.int32)
>>> print(nonzero_num)
[[3]]
>>> # case 2: all value is default.
>>> nonzero_num = x.count_nonzero()
>>> print(nonzero_num)
3
>>> # case 3: axis value was specified 0.
>>> nonzero_num = x.count_nonzero(axis=[0,])
>>> print(nonzero_num)
[1 2 0]
>>> # case 4: axis value was specified 1.
>>> nonzero_num = x.count_nonzero(axis=[1,])
>>> print(nonzero_num)
[1 2]
>>> # case 5: keep_dims value was specified.
>>> nonzero_num = x.count_nonzero(keep_dims=True)
>>> print(nonzero_num)
[[3]]
>>> # case 6: keep_dims and axis value was specified.
>>> nonzero_num = x.count_nonzero(axis=[0,], keep_dims=True)
>>> print(nonzero_num)
[[1 2 0]]
```

mindspore.Tensor.cov

Tensor.**cov**(*, *correction=1*, *fweights=None*, *aweights=None*)

For details, please refer to *mindspore.ops.cov()*.

mindspore.Tensor.cross

Tensor.**cross**(*other*, *dim=None*)

For details, please refer to *mindspore.ops.cross()*.

mindspore.Tensor.cummax

Tensor.**cummax**(*axis*)

For details, please refer to *mindspore.ops.cummax()*.

mindspore.Tensor.cummin

Tensor.**cummin**(*axis*)
For details, please refer to *mindspore.ops.cummin()*.

mindspore.Tensor.cumprod

Tensor.**cumprod**(*dim*, *dtype=None*)
For details, please refer to *mindspore.ops.cumprod()*.

mindspore.Tensor.cumsum

Tensor.**cumsum**(*dim*, *, *dtype=None*) → *Tensor*
Computes the cumulative sum of self Tensor along *dim*.

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Parameters

dim (*int*) −Dim along which the cumulative sum is computed.

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The desired dtype of returned Tensor. If specified, the self Tensor will be cast to *dtype* before the computation. This is useful for preventing overflows. If not specified, stay the same as original Tensor. Default: None .

Returns

Tensor, the shape of the output Tensor is consistent with the self Tensor's.

Raises

ValueError −If the *dim* is out of range.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[3, 4, 6, 10], [1, 6, 7, 9], [4, 3, 8, 7], [1, 3, 7, 9]]).astype(np.
↪float32))
>>> # case 1: along the dim 0
>>> y = x.cumsum(dim=0)
>>> print(y)
[[ 3.  4.  6. 10.]
 [ 4. 10. 13. 19.]
 [ 8. 13. 21. 26.]
 [ 9. 16. 28. 35.]]
>>> # case 2: along the dim 1
>>> y = x.cumsum(dim=1)
>>> print(y)
```

(continues on next page)

(continued from previous page)

```
[[ 3.  7. 13. 23.]
 [ 1.  7. 14. 23.]
 [ 4.  7. 15. 22.]
 [ 1.  4. 11. 20.]]
```

`Tensor.`**`cumsum`**(*axis=None*, *dtype=None*) → *Tensor*

Computes the cumulative sum of self Tensor along *axis*.

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Note: On Ascend, the dtype of *self* only supports :int8, uint8, int32, float16 or float32 in case of static shape. For the case of dynamic shape, the dtype of *self* only supports int32, float16 or float32.

Parameters

- **`axis`** (*int*) −Axis along which the cumulative sum is computed.
- **`dtype`** (*mindspore.dtype*, optional) −The desired dtype of returned Tensor. If specified, the self Tensor will be cast to *dtype* before the computation. This is useful for preventing overflows. If not specified, stay the same as original Tensor. Default: `None` .

Returns

Tensor, the shape of the output Tensor is consistent with the self Tensor's.

Raises

`ValueError` −If the axis is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[3, 4, 6, 10], [1, 6, 7, 9], [4, 3, 8, 7], [1, 3, 7, 9]]).astype(np.
↪float32))
>>> # case 1: along the axis 0
>>> y = x.cumsum(axis=0)
>>> print(y)
[[ 3.  4.  6. 10.]
 [ 4. 10. 13. 19.]
 [ 8. 13. 21. 26.]
 [ 9. 16. 28. 35.]]
>>> # case 2: along the axis 1
>>> y = x.cumsum(axis=1)
>>> print(y)
[[ 3.  7. 13. 23.]
 [ 1.  7. 14. 23.]
 [ 4.  7. 15. 22.]
 [ 1.  4. 11. 20.]]
```

mindspore.Tensor.deg2rad

Tensor.**deg2rad**()
For details, please refer to *mindspore.ops.deg2rad()*.

mindspore.Tensor.diag

Tensor.**diag**() → *Tensor*
For details, please refer to *mindspore.ops.diag()*.

Tensor.**diag**(*diagonal=0*) → *Tensor*

If input is a vector (1-D tensor), then returns a 2-D square tensor with the elements of input as the diagonal.

If input is a matrix (2-D tensor), then returns a 1-D tensor with the diagonal elements of input.

The argument diagonal controls which diagonal to consider:

- If *diagonal* = 0, it is the main diagonal.
- If *diagonal* > 0, it is above the main diagonal.
- If *diagonal* < 0, it is below the main diagonal.

Warning:

- This is an experimental API that is subject to change or deletion.
- The graph mode and CPU/GPU backends do not support non-zero values for the diagonal parameter.

Parameters

diagonal (*int, optional*) – the diagonal to consider. Default: 0.

Returns

- If *input* shape is (x_0): then output shape is $(x_0 + |diagonal|, x_0 + |diagonal|)$ 2-D Tensor.
- If *input* shape is (x_0, x_1): then output shape is main diagonal to move $(|diagonal|)$ elements remains elements' length 1-D Tensor.

Return type

Tensor, has the same dtype as the *input*, its shape is up to *diagonal*

Raises

ValueError – If shape of *input* is not 1-D and 2-D.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor
>>> input = Tensor([1, 2, 3, 4]).astype('int32')
>>> output = input.diag()
>>> print(output)
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

mindspore.Tensor.diagflat

Tensor.**diagflat**(*offset=0*)
For details, please refer to *mindspore.ops.diagflat()*.

mindspore.Tensor.diagonal

Tensor.**diagonal**(*offset=0*, *axis1=0*, *axis2=1*)
For details, please refer to *mindspore.ops.diagonal()*. The parameter *axis1* of the current interface is the same as the parameter *dim1* of the reference interface, the parameter *axis2* of the current interface is the same as the parameter *dim2* of the reference interface.

mindspore.Tensor.diagonal_scatter

Tensor.**diagonal_scatter**(*src*, *offset=0*, *dim1=0*, *dim2=1*)
For details, please refer to *mindspore.ops.diagonal_scatter()*.

mindspore.Tensor.diff

Tensor.**diff**(*n=1*, *axis=- 1*, *prepend=None*, *append=None*)
For details, please refer to *mindspore.ops.diff()*.

mindspore.Tensor.digamma

Tensor.**digamma**()
For details, please refer to *mindspore.ops.digamma()*.

mindspore.Tensor.div

Tensor.**div**(*other*, *, *rounding_mode=None*) → *Tensor*
For details, please refer to *mindspore.ops.div()*.

mindspore.Tensor.div_

Tensor.**div_**(*other*, *, *rounding_mode=None*) → *Tensor*

In-place version of *mindspore.Tensor.div()*.

mindspore.Tensor.divide

Tensor.**divide**(*other*, *, *rounding_mode=None*) → *Tensor*

Alias for *mindspore.Tensor.div()*.

mindspore.Tensor.dot

Tensor.**dot**(*other*) → *Tensor*

Computes the dot product of two 1D tensor.

Parameters

other (Tensor) – The input in the dot product, must be 1D.

Returns

Tensor, the shape is [] and the data type is same as *self*.

Raises

- **TypeError** – If *other* is not tensor.
- **RuntimeError** – If dtypes of *self* and *other* are not same.
- **RuntimeError** – If shapes of *self* and *other* are not same.
- **RuntimeError** – If shapes of *self* and *other* are not 1D.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> input = Tensor([2.0, 3.0], mindspore.float32)
>>> other = Tensor([2.0, 1.0], mindspore.float32)
>>> output = Tensor.dot(input, other)  # input.dot(other)
>>> print(output)
[7.        ]
>>> print(output.dtype)
Float32
```

mindspore.Tensor.double

Tensor.**double**()

Converts input tensor dtype to *float64*.

Returns

Tensor, converted to the *float64* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.int32)
>>> output = input_x.double()
>>> print(output.dtype)
Float64
```

mindspore.Tensor.dsplit

Tensor.**dsplit**(*indices_or_sections*)

For details, please refer to *mindspore.ops.dsplit()*.

mindspore.Tensor.dtype

property Tensor.**dtype**

Return the dtype of the tensor (*mindspore.dtype*).

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([1, 2], dtype=np.float32))
>>> print(x.dtype)
Float32
```

mindspore.Tensor.eigvals

Tensor.**eigvals**()
For details, please refer to *mindspore.ops.eigvals()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.eq

Tensor.**eq**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.eq()*.

mindspore.Tensor.equal

Tensor.**equal**(*other*)
For details, please refer to *mindspore.ops.equal()*.

mindspore.Tensor.erf

Tensor.**erf**() → *Tensor*
For details, please refer to *mindspore.ops.erf()*.

mindspore.Tensor.erfc

Tensor.**erfc**() → *Tensor*
For details, please refer to *mindspore.ops.erfc()*.

mindspore.Tensor.erfinv

Tensor.**erfinv**()
For details, please refer to *mindspore.ops.erfinv()*.

mindspore.Tensor.exp

Tensor.**exp**() → *Tensor*
For details, please refer to *mindspore.ops.exp()*.

mindspore.Tensor.exp_

Tensor.**exp_**() → *Tensor*

Inplace version of *mindspore.Tensor.exp()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.expand

Tensor.**expand**(*size*)

For details, please refer to *mindspore.ops.broadcast_to()*. The parameter *size* of the current interface is the same as the parameter *shape* of the reference interface.

mindspore.Tensor.expand_as

Tensor.**expand_as**(*other*) → *Tensor*

Expand the shape of the input tensor to be the same as the another input tensor. The dim of the input shape must be smaller than or equal to that of another and the broadcast rules must be met.

Parameters

other (Tensor) –The target Tensor. It's shape is the target shape that input tensor need to be expanded.

Returns

Tensor, with the given shape of *other* and the same data type as *self*.

Raises

- **TypeError** –If *other* is not a tensor.
- **ValueError** –If the shapes of *other* and *self* are incompatible.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [1, 2, 3]]).astype(np.float32))
>>> other = Tensor(np.array([[1, 1, 1], [1, 1, 1]]).astype(np.float32))
>>> output = x.expand_as(other)
>>> print(output)
[[1. 2. 3.]
 [1. 2. 3.]]
```

Tensor.**expand_as**(*x*) → *Tensor*

Expand the dimension of input tensor to the dimension of target tensor.

Parameters

x (Tensor) –The target tensor. The shape of the target tensor must obey the broadcasting rule.

Returns

Tensor, has the same dimension as target tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> input = Tensor([1, 2, 3], dtype=mstype.float32)
>>> x = Tensor(np.ones((2, 3)), dtype=mstype.float32)
>>> output = input.expand_as(x=x)
>>> print(output)
[[1. 2. 3.]
 [1. 2. 3.]]
```

mindspore.Tensor.expand_dims

`Tensor.`**`expand_dims`**(*axis*)
For details, please refer to *mindspore.ops.expand_dims()*.

mindspore.Tensor.expm1

`Tensor.`**`expm1`**`()` → *Tensor*
For details, please refer to *mindspore.ops.expm1()*.

mindspore.Tensor.exponential_

`Tensor.`**`exponential_`**(*lambd=1*, *, *generator=None*)
Fills *self* tensor with elements drawn from the exponential distribution:

$$f(x) = \lambda \exp(-\lambda x)$$

Warning:

- It is only supported on Atlas A2 Training Series Products.
- This is an experimental API that is subject to change or deletion.

Parameters

`lambd` (*float, optional*) – Parameters of exponential distribution. Default: `1`.

Keyword Arguments

`generator` (*Generator, optional*) – a pseudorandom number generator. Default: `None` .

Returns

Tensor, with same shape and same data type with input.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> x = mindspore.Tensor([1, 2, 3.0])
>>> out = x.exponential_(2)
>>> print(out.shape)
(3,)
```

mindspore.Tensor.fill_

`Tensor.`**`fill_`**`(`*value*`)` → *Tensor*

Fills *self* tensor with the specified *value* .

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

value (*Union[Tensor, number.Number, bool]*) −Value to fill the *self* .

Returns

Tensor.

Raises

- **RunTimeError** −The data type of *self* or *value* is not supported.
- **RunTimeError** −When the *value* is Tensor, it should be 0-D Tensor or 1-D Tensor with shape=[1].

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> x = ops.zeros((3, 3))
>>> print(x)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
>>> output = x.fill_(1.0)
>>> print(output)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

(continues on next page)

(continued from previous page)

```
>>> print(x)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

mindspore.Tensor.fill_diagonal

Tensor.**fill_diagonal**(*fill_value*, *wrap=False*)

Fills the main diagonal of a Tensor with a specified value and returns the result. The input has at least 2 dimensions, and all dimensions of input must be equal in length when the dimension of input is greater than 2.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **fill_value** (*float*) –The value to fill with the diagonal of *self*.
- **wrap** (*bool, optional*) –Controls whether the diagonal elements continue onto the remaining rows in case of a tall matrix(a matrix has more rows than columns). Default: False.

Returns

- **y** (Tensor) - Tensor, has the same shape and data type as *self*.

Raises

- **TypeError** –If data type of *self* is not one of the following: float32, int32, int64.
- **ValueError** –If the dimension of *self* is not greater than 1.
- **ValueError** –If the size of each dimension is not equal, when the dimension is greater than 2.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.ones((6, 3)), mindspore.float32)
>>> output = x.fill_diagonal(5.0, wrap=True)
>>> print(output)
[[5. 1. 1.]
 [1. 5. 1.]
 [1. 1. 5.]
 [1. 1. 1.]
 [5. 1. 1.]
 [1. 5. 1.]]
```

mindspore.Tensor.flatten

Tensor.**flatten**(*start_dim=0*, *end_dim=- 1*) → *Tensor*

Flatten a tensor along dimensions from *start_dim* to *end_dim*.

Parameters

- **start_dim** (*int, optional*) – The first dimension to flatten. Default: `0` .
- **end_dim** (*int, optional*) – The last dimension to flatten. Default: `-1` .

Returns

Tensor. If no dimensions are flattened, returns the original *self*, otherwise return the flattened Tensor. If *self* is a 0-dimensional Tensor, a 1-dimensional Tensor will be returned.

Raises

- **TypeError** – If *start_dim* or *end_dim* is not int.
- **ValueError** – If *start_dim* is greater than *end_dim* after canonicalized.
- **ValueError** – If *start_dim* or *end_dim* is not in range of [-self.dim, self.dim-1].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones(shape=[1, 2, 3, 4]), mindspore.float32)
>>> output = input_x.flatten(0, -1)
>>> print(output.shape)
(24,)
```

Tensor.**flatten**(*order='C'*, ***, *start_dim=0*, *end_dim=- 1*) → *Tensor*

Flatten a tensor along dimensions from *start_dim* to *start_dim*.

Parameters

order (*str, optional*) – Only `'C'` and `'F'` are supported. `'C'` means to flatten in row-major (C-style) order. `'F'` means to flatten in column-major (Fortran-style) order. Default: `'C'` .

Keyword Arguments

- **start_dim** (*int, optional*) – The first dimension to flatten. Default: `0` .
- **end_dim** (*int, optional*) – The last dimension to flatten. Default: `-1` .

Returns

Tensor. If no dimensions are flattened, returns the original *self*, otherwise return the flattened Tensor. If *self* is a 0-dimensional Tensor, a 1-dimensional Tensor will be returned.

Raises

- **TypeError** – If *order* is not string type.
- **ValueError** – If *order* is string type, but not `'C'` or `'F'`.

- **TypeError** – If *start_dim* or *end_dim* is not int.
- **ValueError** – If *start_dim* is greater than *end_dim* after canonicalized.
- **ValueError** – If *start_dim* or *end_dim* is not in range of [-self.dim, self.dim-1].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones(shape=[1, 2, 3, 4]), mindspore.float32)
>>> output = input_x.flatten(order='C')
>>> print(output.shape)
(24,)
```

mindspore.Tensor.flip

Tensor.**flip**(*dims*)

For details, please refer to *mindspore.ops.flip()*.

mindspore.Tensor.fliplr

Tensor.**fliplr**()

For details, please refer to *mindspore.ops.fliplr()*.

mindspore.Tensor.flipud

Tensor.**flipud**()

For details, please refer to *mindspore.ops.flipud()*.

mindspore.Tensor.float

Tensor.**float**()

Converts input tensor dtype to *float32*.

Returns

Tensor, converted to the *float32* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.int32)
>>> output = input_x.float()
>>> print(output.dtype)
Float32
```

mindspore.Tensor.float_power

Tensor.**float_power**(*other*)

For details, please refer to *mindspore.ops.float_power()*.

mindspore.Tensor.floor

Tensor.**floor**() → *Tensor*

For details, please refer to *mindspore.ops.floor()*.

mindspore.Tensor.floor_

Tensor.**floor_**()

In-place version of *mindspore.Tensor.floor()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.floor_divide

Tensor.**floor_divide**(*other*) → *Tensor*

Divides the self tensor by the other input tensor element-wise and round down to the closest integer.

self and *other* comply with the implicit type conversion rules to make the data types consistent. Inputs must be two tensors or one tensor and one scalar. When the *self* and *other* are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. When the *self* and *other* are one tensor and one scalar, the scalar could only be a constant.

$$out_i = \text{floor}(\frac{self_i}{other_i})$$

where the $floor$ indicates the Floor operator. For more details, please refer to the *mindspore.mint.floor* operator.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

other (*Union[Tensor, Number, bool]*) – The other input is a number or a bool or a tensor whose data type is number or bool.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits between *self* and *other*.

Raises

TypeError – If *self* and *other* are not the following: Tensor, number.Number or bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> import numpy as np
>>> input = Tensor(np.array([2, 4, -1]), mindspore.int32)
>>> other = Tensor(np.array([3, 3, 3]), mindspore.int32)
>>> output = input.floor_divide(other)
>>> print(output)
[ 0  1 -1]
>>> input = Tensor(2.0, mindspore.float32)
>>> other = Tensor(2.0, mindspore.float32)
>>> output = input.floor_divide(other)
>>> print(output)
1.0
```

mindspore.Tensor.floor_divide_

Tensor.**floor_divide_**(*other*) → *Tensor*

Divides the self tensor by the other tensor element-wise and round down to the closest integer.

$$out_i = \text{floor}(\frac{self_i}{other_i})$$

where the $floor$ indicates the Floor operator. For more details, please refer to the *mindspore.mint.floor* operator.

Warning: This is an experimental API that is subject to change or deletion.

Note: When *self* and *other* have different shapes, *other* should be able to broadcast to *self*.

Parameters

other (*Union[Tensor, Number, bool]*) – The other input is a number or a bool or a tensor whose data type is number or bool.

Returns

Tensor, the shape is the same as *self* , and the data type is the same as *self* .

Raises

- **TypeError** – If *other* is not one of the following: Tensor, number.Number or bool.

- **RuntimeError** – If *other* cannot be broadcast to *self*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([2, 4, -1]), mindspore.int32)
>>> other = Tensor(np.array([3, 3, 3]), mindspore.int32)
>>> output = x.floor_divide_(other)
>>> print(output)
[ 0  1 -1]
>>> print(x)
[ 0  1 -1]
```

mindspore.Tensor.flush_from_cache

`Tensor.`**`flush_from_cache`**`()`

Flush cache data to host if tensor is cache enable.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([1, 2], dtype=np.float32))
>>> y = x.flush_from_cache()
>>> print(y)
None
```

mindspore.Tensor.fmax

`Tensor.`**`fmax`**`(`*other*`)`

For details, please refer to *mindspore.ops.fmax()*.

mindspore.Tensor.fmod

`Tensor.`**`fmod`**`(`*other*`)` → *Tensor*

For details, please refer to *mindspore.ops.fmod()*.

mindspore.Tensor.fold

Tensor.**fold**(*output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)
For details, please refer to *mindspore.ops.fold()*.

mindspore.Tensor.frac

Tensor.**frac**() → *Tensor*
For details, please refer to *mindspore.ops.frac()*.

mindspore.Tensor.from_numpy

static Tensor.**from_numpy**(*array*)
Convert numpy array to Tensor. If the data is not C contiguous, the data will be copied to C contiguous to construct the tensor. Otherwise, The tensor will be constructed using this numpy array without copy.

Parameters

array (*numpy.array*) –The input array.

Returns

Tensor, has the same data type as input array.

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = np.array([1, 2])
>>> output = Tensor.from_numpy(x)
>>> print(output)
[1 2]
```

mindspore.Tensor.gather

Tensor.**gather**(*dim*, *index*) → *Tensor*
Gather data from a tensor by indices.

$$output[(i_0, i_1, ..., i_{dim}, i_{dim+1}, ..., i_n)] = input[(i_0, i_1, ..., index[(i_0, i_1, ..., i_{dim}, i_{dim+1}, ..., i_n)], i_{dim+1}, ..., i_n)]$$

Warning: On Ascend, the behavior is unpredictable in the following cases:

- the value of *index* is not in the range *[-self.shape[dim], self.shape[dim])* in forward;
- the value of *index* is not in the range *[0, self.shape[dim])* in backward.

Parameters

- **dim** (*int*) –the axis to index along, must be in range *[-self.rank, self.rank)*.
- **index** (Tensor) –The index tensor, with int32 or int64 data type. A valid *index* should be:

- $index.rank == self.rank$;
- for $axis != dim$, $index.shape[axis] <= self.shape[axis]$;
- the value of *index* is in range $[-self.shape[dim], self.shape[dim])$.

Returns

Tensor, has the same type as *self* and the same shape as *index*.

Raises

- **ValueError** – If the shape of *index* is illegal.
- **ValueError** – If *dim* is not in $[-self.rank, self.rank)$.
- **ValueError** – If the value of *index* is out of the valid range.
- **TypeError** – If the type of *index* is illegal.

Supported Platforms:

`Ascend GPU CPU`

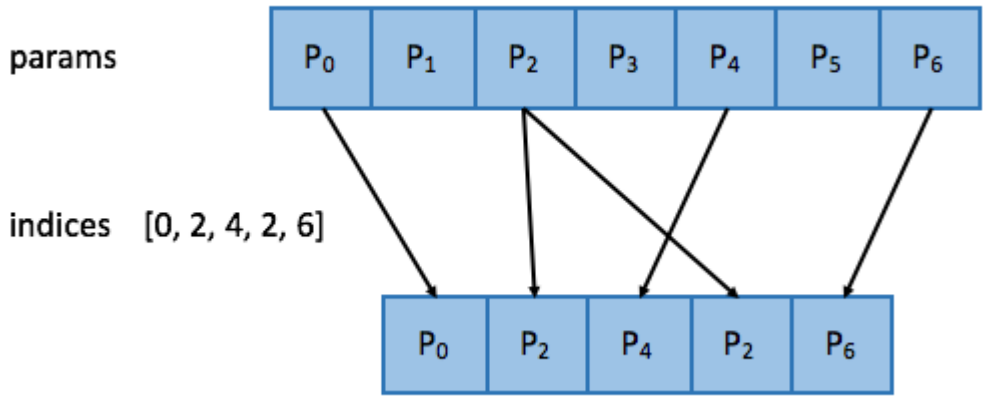
Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> index = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> output = input.gather(1, index)
>>> print(output)
[[-0.1 -0.1]
 [0.5   0.5]]
```

`Tensor.`**`gather`**(*input_indices*, *axis*, *batch_dims=0*) → *Tensor*

Returns the slice of the input tensor corresponding to the elements of *input_indices* on the specified *axis*.

The following figure shows the calculation process of Gather commonly:



where params represents the input *input_params*, and indices represents the index to be sliced *input_indices*.

Note:

- The value of input_indices must be in the range of *[0, input_param.shape[axis])*. On CPU and GPU, an error is raised if an out of bound indice is found. On Ascend, the results may be undefined.
- The data type of self cannot be bool_ on Ascend platform currently.

Parameters

- **input_indices** (Tensor) –Index tensor to be sliced, the shape of tensor is $(y_1, y_2, ..., y_S)$. Specifies the indices of elements of the original Tensor. The data type can be int32 or int64.
- **axis** (*Union(int, Tensor[int])*) –Specifies the dimension index to gather indices. It must be greater than or equal to *batch_dims*. When *axis* is a Tensor, the size must be 1.
- **batch_dims** (*int, optional*) –Specifies the number of batch dimensions. It must be less than or euqal to the rank of *input_indices*. Default: 0 .

Returns

Tensor, the shape of tensor is $input\_params.shape[:axis] + input\_indices.shape[batch\_dims:] + input\_params.shape[axis+1:]$.

Raises

- **TypeError** –If *axis* is not an int or Tensor.
- **ValueError** –If *axis* is a Tensor and its size is not 1.
- **TypeError** –If *self* is not a tensor.
- **TypeError** –If *input_indices* is not a tensor of type int.
- **RuntimeError** –If *input_indices* is out of range $[0, input_param.shape[axis])$ on CPU or GPU.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> # case1: input_indices is a Tensor with shape (5, ).
>>> input_params = Tensor(np.array([1, 2, 3, 4, 5, 6, 7]), mindspore.float32)
>>> input_indices = Tensor(np.array([0, 2, 4, 2, 6]), mindspore.int32)
>>> axis = 0
>>> output = input_params.gather(input_indices=input_indices, axis=axis)
>>> print(output)
[1. 3. 5. 3. 7.]
>>> # case2: input_indices is a Tensor with shape (2, 2). When the input_params has one␣
↪dimension,
>>> # the output shape is equal to the input_indices shape.
>>> input_indices = Tensor(np.array([[0, 2], [2, 6]]), mindspore.int32)
>>> axis = 0
>>> output = input_params.gather(input_indices=input_indices, axis=axis)
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[1. 3.]
 [3. 7.]]
>>> # case3: input_indices is a Tensor with shape (2, ) and
>>> # input_params is a Tensor with shape (3, 4) and axis is 0.
>>> input_params = Tensor(np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]), mindspore.
↪float32)
>>> input_indices = Tensor(np.array([0, 2]), mindspore.int32)
>>> axis = 0
>>> output = input_params.gather(input_indices=input_indices, axis=axis)
>>> print(output)
[[ 1.  2.  3.  4.]
 [ 9. 10. 11. 12.]]
>>> # case4: input_indices is a Tensor with shape (2, ) and
>>> # input_params is a Tensor with shape (3, 4) and axis is 1, batch_dims is 1.
>>> input_params = Tensor(np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]), mindspore.
↪float32)
>>> input_indices = Tensor(np.array([0, 2, 1]), mindspore.int32)
>>> axis = 1
>>> batch_dims = 1
>>> output = input_params.gather(input_indices, axis, batch_dims)
>>> print(output)
[ 1.  7. 10.]
```

mindspore.Tensor.gather_elements

Tensor.**gather_elements**(*dim*, *index*)

For details, please refer to *mindspore.ops.gather_elements()*.

mindspore.Tensor.gather_nd

Tensor.**gather_nd**(*indices*)

For details, please refer to *mindspore.ops.gather_nd()*.

mindspore.Tensor.gcd

Tensor.**gcd**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.gcd()*.

mindspore.Tensor.ge

Tensor.**ge**(*other*) → *Tensor*

Alias for *mindspore.Tensor.greater_equal()*.

mindspore.Tensor.geqrf

Tensor.**geqrf**()
For details, please refer to *mindspore.ops.geqrf()*.

mindspore.Tensor.ger

Tensor.**ger**(*vec2*)
For details, please refer to *mindspore.ops.ger()*.

mindspore.Tensor.greater

Tensor.**greater**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.greater()*.

mindspore.Tensor.greater_equal

Tensor.**greater_equal**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.greater_equal()*.

mindspore.Tensor.gt

Tensor.**gt**(*other*) → *Tensor*
For details, please refer to :func:'mindspore.Tensor.greater'.

mindspore.Tensor.H

property Tensor.**H**
Returns a view of a matrix (2-D tensor) conjugated and transposed. x.H is equivalent to *mindspore.Tensor.swapaxes(0, 1).conj()* for complex matrices and *mindspore.Tensor.swapaxes(0, 1)* for real matrices.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.H
>>> print(output)
[[1 3]
[2 4]]
```

mindspore.Tensor.half

Tensor.**half**()

Converts input tensor dtype to *float16*.

Returns

Tensor, converted to the *float16* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.int32)
>>> output = input_x.half()
>>> print(output.dtype)
Float16
```

mindspore.Tensor.hardshrink

Tensor.**hardshrink**(*lambd=0.5*) → *Tensor*

For details, please refer to *mindspore.ops.hardshrink()*.

mindspore.Tensor.has_init

property Tensor.**has_init**

Whether tensor is initialized.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.has_init
>>> print(output)
False
```

mindspore.Tensor.heaviside

Tensor.**heaviside**(*values*)
For details, please refer to *mindspore.ops.heaviside()*.

mindspore.Tensor.histc

Tensor.**histc**(*bins=100*, *min=0*, *max=0*) → *Tensor*
For details, please refer to *mindspore.ops.histc()*.

Supported Platforms:

Ascend GPU CPU

mindspore.Tensor.hsplit

Tensor.**hsplit**(*indices_or_sections*)
For details, please refer to *mindspore.ops.hsplit()*.

mindspore.Tensor.hypot

Tensor.**hypot**(*other*)
For details, please refer to *mindspore.ops.hypot()*.

mindspore.Tensor.i0

Tensor.**i0**()
For details, please refer to *mindspore.ops.bessel_i0()*.

mindspore.Tensor.igamma

Tensor.**igamma**(*other*)
For details, please refer to *mindspore.ops.igamma()*.

mindspore.Tensor.igammac

Tensor.**igammac**(*other*)
For details, please refer to *mindspore.ops.igammac()*.

mindspore.Tensor.imag

Tensor.**imag**()
For details, please refer to *mindspore.ops.imag()*.

mindspore.Tensor.index_add

Tensor.**index_add**(*indices*, *y*, *axis*, *use_lock=True*, *check_index_bound=True*) → *Tensor*
Adds tensor *y* to specified axis and indices of tensor *self*. The axis should be in [-len(self.dim), len(self.dim) - 1], and indices should be in [0, the size of *self* - 1] at the axis dimension.

Parameters

- **indices** (Tensor) –Add the value of *self* and *y* along the dimension of the *axis* according to the specified index value, with data type int32. The *indices* must be 1D with the same size as the size of *y* in the *axis* dimension. The values of *indices* should be in [0, b), where the b is the size of *self* in the *axis* dimension.
- **y** (Tensor) –The input tensor with the value to add.
- **axis** (*int*) –The dimension along which to index.
- **use_lock** (*bool, optional*) –Whether to enable a lock to protect the updating process of variable tensors. If True , when updating the value of *self*, this process will be protected by a lock by using atomic operation. If False , the result may be unpredictable. Default: True .
- **check_index_bound** (*bool, optional*) –If True , check indices boundary. If False , don't check indices boundary. Default: True .

Returns

Tensor, has the same shape and dtype as *self*.

Raises

- **TypeError** –If neither *indices* nor *y* is a Tensor.
- **ValueError** –If axis is out of the range of *self* shape.
- **ValueError** –If *self* rank is not the same as *y* rank.
- **ValueError** –If shape of *indices* is not 1D or size of *indices* is not equal to dimension of y[axis].
- **ValueError** –If *y*'s shape is not the same as *self* except the *axis* th dimension.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> indices = Tensor(np.array([0, 2]), mindspore.int32)
>>> y = Tensor(np.array([[0.5, 1.0], [1.0, 1.5], [2.0, 2.5]]), mindspore.float32)
>>> output = x.index_add(indices, y, axis = 1)
>>> print(output)
[[ 1.5  2.   4. ]
```

(continues on next page)

(continued from previous page)

```
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
```

`Tensor.`**`index_add`**(*dim*, *index*, *source*, *, *alpha=1*) → *Tensor*

For details, please refer to `mindspore.ops.index_add()`. The corresponding relationships between the parameters of *Tensor.index_add* and `mindspore.ops.index_add()` are as follows: *dim* -> *axis*, *index* -> *indices*, *source* * *alpha* -> *y*.

mindspore.Tensor.index_add_

`Tensor.`**`index_add_`**(*dim*, *index*, *source*, *, *alpha=1*)

Accumulate the elements of *alpha* times *source* into the *self* by adding to the index in the order given in *index*. For example, if *dim* == *0*, *index[i]* == *j*, and *alpha* = *-1*, then the *i* th row of *source* is subtracted from the *j* th row of *self* . The *dim* th dimension of *source* must have the same size as the length of *index* , and all other dimensions must match *self*, or an error will be raised. For a 3-D tensor the output is defined as follows:

$$
\begin{aligned}
self[index[i],\ :,\ :] \mathrel{+}= alpha * src[i,\ :,\ :] \quad & \#if\ dim == 0 \\
self[:,\ index[i],\ :] \mathrel{+}= alpha * src[:,\ i,\ :] \quad & \#if\ dim == 1 \\
self[:,\ :,\ index[i]] \mathrel{+}= alpha * src[:,\ :,\ i] \quad & \#if\ dim == 2
\end{aligned}
$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **`dim`** (*`int`*) −The dimension along which to index.
- **`index`** (`Tensor`) −Add the value of *self* and *source* along the dimension of the *dim* according to the specified index value, with data type int32. The *index* must be 1D with the same size as the size of *source* in the *dim* dimension. The values of *index* should be in [0, b), where the b is the size of *self* in the *dim* dimension.
- **`source`** (`Tensor`) −The input tensor with the value to add. Must have same data type as *self*. The shape must be the same as *self* except the *dim* th dimension.

Keyword Arguments

`alpha` (*`number, optional`*) −The scalar multiplier for source. Default: `1`.

Returns

Tensor, has the same shape and dtype as *self*.

Raises

- **`TypeError`** −If neither *index* nor *source* is a Tensor.
- **`ValueError`** −If dim is out of *self* rank's range.
- **`ValueError`** −If *self* rank is not the same as *source* rank.
- **`ValueError`** −If shape of *index* is not 1D or size of *index* is not equal to dimension of *source[dim]*.
- **`ValueError`** −If *source*'s shape is not the same as *self* except the *dim* th dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> index = Tensor(np.array([0, 2]), mindspore.int32)
>>> y = Tensor(np.array([[0.5, 1.0], [1.0, 1.5], [2.0, 2.5]]), mindspore.float32)
>>> output = x.index_add_(1, index, y, alpha=1)
>>> print(output)
[[ 1.5  2.   4. ]
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
>>> print(x)
[[ 1.5  2.   4. ]
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
```

mindspore.Tensor.index_fill

Tensor.**index_fill**(*axis*, *index*, *value*)

For details, please refer to *mindspore.ops.index_fill()*.

mindspore.Tensor.index_put

Tensor.**index_put**(*indices*, *values*, *accumulate=False*)

Based on the indices in *indices*, replace the corresponding elements in Tensor *self* with the values in *values*. Outplace version of *mindspore.Tensor.index_put_()* 。

> **Warning:** The behavior is unpredictable in the following scenario:
>
> - If *accumulate* is *False* and *indices* contains duplicate elements.

Parameters

- **indices** (*tuple[Tensor], list[Tensor]*) – the indices of type int32 or int64, used to index into the *self*. The rank of tensors in indices should be 1-D, size of indices should <= *self.rank* and the tensors in indices should be broadcastable.
- **values** (Tensor) – 1-D Tensor with the same type as *self*. *values* should be broadcastable with size 1.
- **accumulate** (*bool, optional*) – If *accumulate* is *True*, the elements in *values* will be added to *self*, otherwise the elements in *values* will replace the corresponding elements in the *self*. Default: False.

Returns

Tensor, with the same type and shape as the "self Tensor".

Raises

- **TypeError** – If the dtype of the *self* is not equal to the dtype of *values*.
- **TypeError** – If the dtype of *indices* is not tuple[Tensor], list[Tensor].
- **TypeError** – If the dtype of tensors in *indices* are not int32 or int64.

- **TypeError** –If the dtype of tensors in *indices* are inconsistent.
- **TypeError** –If the dtype of *accumulate* is not bool.
- **ValueError** –If rank(*values*) is not 1-D.
- **ValueError** –If size(*values*) is not 1 or max size of the tensors in *indices* when rank(*self*) == size(*indices*).
- **ValueError** –If size(*values*) is not 1 or *self*.shape[-1] when rank(*self*) > size(*indices*).
- **ValueError** –If the rank of tensors in *indices* is not 1-D.
- **ValueError** –If the tensors in *indices* is not be broadcastable.
- **ValueError** –If size(*indices*) > rank(*self*).

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]).astype(np.int32))
>>> values = Tensor(np.array([3]).astype(np.int32))
>>> indices = [Tensor(np.array([0, 1, 1]).astype(np.int32)), Tensor(np.array([1, 2, 1]).
↪astype(np.int32))]
>>> accumulate = True
>>> output = x.index_put(indices, values, accumulate)
>>> print(output)
[[1 5 3]
[4 8 9]]
```

mindspore.Tensor.index_put_

`Tensor.`**`index_put_`**(*indices*, *values*, *accumulate=False*)

Based on the indices in *indices*, replace the corresponding elements in Tensor *self* with the values in *values*. The expression *Tensor.index_put_(indices, values)* is equivalent to *tensor[indices] = values*. Update and return *self*.

> **Warning:** The behavior is unpredictable in the following scenario:
>
> - If *accumulate* is *False* and *indices* contains duplicate elements.

Parameters

- **indices** (*tuple[Tensor], list[Tensor]*) –the indices of type is bool, uint8, int32 or int64, used to index into the *self*. The size of indices should <= the rank of *self* and the tensors in indices should be broadcastable.
- **values** (`Tensor`) –Tensor with the same type as *self*. If size == 1, it will be broadcastable.
- **accumulate** (*bool, optional*) –If *accumulate* is *True*, the elements in *values* will be added to *self*, otherwise the elements in *values* will replace the corresponding elements in the *self*. Default: `False`.

Returns

Tensor *self*.

Raises

- **TypeError** –If the dtype of the *self* is not equal to the dtype of *values*.
- **TypeError** –If the dtype of *indices* is not tuple[Tensor], list[Tensor].
- **TypeError** –If the dtype of tensors in *indices* are not bool, uint8, int32 or int64.
- **TypeError** –If the dtypes of tensors in *indices* are inconsistent.
- **TypeError** –If the dtype of *accumulate* is not bool.
- **ValueError** –If size(*values*) is not 1 or max size of the tensors in *indices* when rank(*self*) == size(*indices*).
- **ValueError** –If size(*values*) is not 1 or *self*.shape[-1] when rank(*self*) > size(*indices*).
- **ValueError** –If the tensors in *indices* is not be broadcastable.
- **ValueError** –If size(*indices*) > rank(*self*).

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]).astype(np.int32))
>>> values = Tensor(np.array([3]).astype(np.int32))
>>> indices = [Tensor(np.array([0, 1, 1]).astype(np.int32)), Tensor(np.array([1, 2, 1]).
↪astype(np.int32))]
>>> accumulate = True
>>> x.index_put_(indices, values, accumulate)
>>> print(x)
[[1 5 3]
 [4 8 9]]
```

mindspore.Tensor.index_select

`Tensor.`**`index_select`**(*axis*, *index*) → *Tensor*

Generates a new Tensor that accesses the values of *self* along the specified *axis* dimension using the indices specified in *index*. The new Tensor has the same number of dimensions as *self*, with the size of the *axis* dimension being equal to the length of *index*, and the size of all other dimensions will be unchanged from the original *self* Tensor.

Note: The value of index must be in the range of *[0, self.shape[axis])*, the result is undefined out of range.

Parameters

- **axis** (*int*) –The dimension to be indexed.
- **index** (Tensor) –A 1-D Tensor with the indices to access in *self* along the specified axis.

Returns

Tensor, has the same dtype as *self* Tensor.

Raises

- **TypeError** –If *index* is not a Tensor.
- **TypeError** –If *axis* is not int number.
- **ValueError** –If the value of *axis* is out the range of *[-self.ndim, self.ndim - 1]*.
- **ValueError** –If the dimension of *index* is not equal to 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> import numpy as np
>>> input = Tensor(np.arange(16).astype(np.float32).reshape(2, 2, 4))
>>> print(input)
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]]
 [[ 8.  9. 10. 11.]
  [12. 13. 14. 15.]]]
>>> index = Tensor([0,], mindspore.int32)
>>> y = input.index_select(1, index)
>>> print(y)
[[[ 0.  1.  2.  3.]]
 [[ 8.  9. 10. 11.]]]
```

`Tensor.`**`index_select`**(*dim*, *index*) → *Tensor*

Generates a new Tensor that accesses the values of *self* along the specified *dim* dimension using the indices specified in *index*. The new Tensor has the same number of dimensions as *self*, with the size of the *dim* dimension being equal to the length of *index*, and the size of all other dimensions will be unchanged from the original *self* Tensor.

Note: The value of index must be in the range of *[0, self.shape[dim])*, the result is undefined out of range.

Parameters

- **dim** (*int*) –The dimension to be indexed.
- **index** (`Tensor`) –A 1-D Tensor with the indices to access in *self* along the specified dim.

Returns

Tensor, has the same dtype as *self* Tensor.

Raises

- **TypeError** –If *index* is not a Tensor.
- **TypeError** –If *dim* is not int number.
- **ValueError** –If the value of *dim* is out the range of *[-self.ndim, self.ndim - 1]*.
- **ValueError** –If the dimension of *index* is not equal to 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> import numpy as np
>>> input = Tensor(np.arange(16).astype(np.float32).reshape(2, 2, 4))
>>> print(input)
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]]
 [[ 8.  9. 10. 11.]
  [12. 13. 14. 15.]]]
>>> index = Tensor([0,], mindspore.int32)
>>> y = input.index_select(1, index)
>>> print(y)
[[[ 0.  1.  2.  3.]]
 [[ 8.  9. 10. 11.]]]
```

mindspore.Tensor.init_data

Tensor.**init_data**(*slice_index=None*, *shape=None*, *opt_shard_group=None*)

Get the tensor format data of this Tensor.

Note: The init_data function can be called once for the same tensor.

Parameters

- **slice_index** (*int*) – Slice index of a parameter's slices. It is used when initialize a slice of a parameter, it guarantees that devices using the same slice can generate the same tensor. Default: None.
- **shape** (*list[int]*) – Shape of the slice, it is used when initialize a slice of the parameter. Default: None.
- **opt_shard_group** (*str*) – Optimizer shard group which is used in auto or semi auto parallel mode to get one shard of a parameter's slice. For more information about optimizer parallel, please refer to: Optimizer Parallel. Default: None.

Returns

Initialized Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore.common.initializer import initializer, Constant
>>> x = initializer(Constant(1), [2, 2], ms.float32)
>>> out = x.init_data()
>>> print(out)
[[1. 1.]
 [1. 1.]]
```

mindspore.Tensor.inner

Tensor.**inner**(*other*)

For details, please refer to *mindspore.ops.inner()*.

mindspore.Tensor.inplace_update

Tensor.**inplace_update**(*v*, *indices*)

For details, please refer to *mindspore.ops.inplace_update()*.

mindspore.Tensor.int

Tensor.**int**()

Converts input tensor dtype to *int32*. If the value in tensor is float or half, the decimal will be discarded.

Returns

Tensor, converted to the *int32* dtype.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.float32)
>>> output = input_x.int()
>>> print(output.dtype)
Int32
```

mindspore.Tensor.inv

Tensor.**inv**()
For details, please refer to *mindspore.ops.inv()*.

mindspore.Tensor.inverse

Tensor.**inverse**() → *Tensor*
For details, please refer to *mindspore.ops.inverse()*.

Supported Platforms:

Ascend GPU CPU

mindspore.Tensor.invert

Tensor.**invert**()
For details, please refer to *mindspore.ops.invert()*.

mindspore.Tensor.isclose

Tensor.**isclose**(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*) → *Tensor*
Returns a tensor of Boolean values indicating whether each element of *input* is "close" to the corresponding element of *other*. Closeness is defined as:

$$|input - other| <= atol + rtol * |other|$$

Parameters

- **other** (Tensor) –Second tensor to compare.
- **rtol** (*float, optional*) –Relative tolerance. Default: 1e-05 .
- **atol** (*float, optional*) –Absolute tolerance. Default: 1e-08 .
- **equal_nan** (*bool, optional*) –If True , then two NaNs will be considered equal. Default: True .

Returns

Tensor, with the same shape as *input* and *other* after broadcasting, its dtype is bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([1.3, 2.1, 3.2, 4.1, 5.1]), mindspore.float16)
>>> other = Tensor(np.array([1.3, 3.3, 2.3, 3.1, 5.1]), mindspore.float16)
>>> output = Tensor.isclose(input, other)
>>> print(output)
[ True False False False  True]
```

Tensor.**isclose**(*x2*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*) → *Tensor*

Returns a new Tensor with boolean elements representing if each element of *input* is "close" to the corresponding element of *x2*. Closeness is defined as:

$$|input - x2| <= atol + rtol * |x2|$$

Parameters

- **x2** (Tensor) – Second tensor to compare. Dtype must be same as *input*.
- **rtol** (*Union[float, int, bool], optional*) – Relative tolerance. Default: `1e-05` .
- **atol** (*Union[float, int, bool], optional*) – Absolute tolerance. Default: `1e-08` .
- **equal_nan** (*bool, optional*) – If `True` , then two NaNs will be considered equal. Default: `False`.

Returns

A bool Tensor, with the shape as broadcasted result of the input *input* and *x2*.

Raises

- **TypeError** – *x2* is not Tensor.
- **TypeError** – *input* or *x2* dtype is not support. Support dtype: float16, float32, float64, int8, int16, int32, int64 and uint8. On Ascend, more dtypes are support: bool and bfloat16.
- **TypeError** – *atol* or *rtol* is not float, int or bool.
- **TypeError** – *equal_nan* is not bool.
- **TypeError** – *input* and *x2* have different dtypes.
- **ValueError** – *input* and *x2* cannot broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([1.3, 2.1, 3.2, 4.1, 5.1]), mindspore.float16)
>>> x2 = Tensor(np.array([1.3, 3.3, 2.3, 3.1, 5.1]), mindspore.float16)
>>> output = Tensor.isclose(input, x2)
>>> print(output)
[ True False False False  True]
```

mindspore.Tensor.isfinite

Tensor.**isfinite**() → *Tensor*

For details, please refer to *mindspore.ops.isfinite()*.

mindspore.Tensor.is_complex

Tensor.**is_complex**()

For details, please refer to: func:'mindspore.ops.is_complex'.

Examples

```
>>> x = mindspore.Tensor([1 + 1j], dtype=mindspore.complex128)
>>> y = mindspore.Tensor([1], dtype=mindspore.int32)
>>> print(x.is_complex())
True
>>> print(y.is_complex())
False
```

mindspore.Tensor.is_contiguous

Tensor.**is_contiguous**()

Determines whether the memory of tensor is contiguous.

Returns

Bool, True if tensor memory is contiguous, False otherwise.

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor([[1, 2, 3], [4, 5, 6]], dtype=ms.float32)
>>> y = ops.transpose(x, (1, 0))
>>> print(y.is_contiguous())
False
```

mindspore.Tensor.is_floating_point

Tensor.**is_floating_point**()

For details, please refer to *mindspore.ops.is_floating_point()*.

mindspore.Tensor.isinf

Tensor.**isinf**() → *Tensor*

For details, please refer to *mindspore.ops.isinf()*.

Supported Platforms:

Ascend CPU GPU

mindspore.Tensor.isnan

Tensor.**isnan**()
For details, please refer to *mindspore.ops.ne()*.

mindspore.Tensor.isneginf

Tensor.**isneginf**() → *Tensor*
For details, please refer to *mindspore.ops.isneginf()*.

mindspore.Tensor.isposinf

Tensor.**isposinf**()
For details, please refer to *mindspore.ops.isposinf()*.

mindspore.Tensor.isreal

Tensor.**isreal**()
For details, please refer to *mindspore.ops.isreal()*.

mindspore.Tensor.is_signed

Tensor.**is_signed**()
Judge whether the data type of tensor is a signed data type.

Returns

Bool. If the dtype of the tensor is a signed data type, return True. Otherwise, return False.

Examples

```
>>> import mindspore as ms
>>> x = ms.Tensor([1, 2, 3], ms.int64)
>>> y = ms.Tensor([1, 2, 3], ms.uint64)
>>> output = x.is_signed()
>>> output2 = y.is_signed()
>>> print(output)
True
>>> print(output2)
False
```

mindspore.Tensor.item

Tensor.**item**()

Return the value of this tensor as standard Python number. This only works for tensors with one element.

Returns

A scalar, type is defined by the dtype of the Tensor.

Raises

- **ValueError** –If the count of value in tensor is more than one.
- **TypeError** –The type of element in tensor is not supported.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> x = Tensor(1.2, ms.float32)
>>> print(x.item())
1.2
```

mindspore.Tensor.itemset

Tensor.**itemset**(**args*)

Insert scalar into a tensor (scalar is cast to tensor's dtype, if possible).

There must be at least 1 argument, and define the last argument as item. Then, tensor.itemset(*args) is equivalent to $Tensor[args] = item$.

Parameters

args (*Union[(numbers.Number), (int/tuple(int), numbers.Number)]*) –The arguments that specify the index and value. If *args* contain one argument (a scalar), it is only used in case tensor is of size 1. If *args* contains two arguments, the last argument is the value to be set and must be a scalar, the first argument specifies a single tensor element location. It is either an int or a tuple.

Returns

A new tensor that doesn't affect the original tensor, with value set by $Tensor[args] = item$.

Raises

- **ValueError** –If the length of the first argument is not equal to self.ndim.
- **IndexError** –If only one argument is provided, and the original Tensor is not scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1,2,3],[4,5,6]], dtype=np.float32))
>>> print(x.itemset((0,1), 4))
[[1. 4. 3.]
[4. 5. 6.]]
>>> print(x)
[[1. 2. 3.]
[4. 5. 6.]]
```

mindspore.Tensor.itemsize

property Tensor.**itemsize**

Return the length of one tensor element in bytes.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.itemsize
>>> print(output)
8
```

mindspore.Tensor.lcm

Tensor.**lcm**(*other*)

For details, please refer to *mindspore.ops.lcm()*.

mindspore.Tensor.ldexp

Tensor.**ldexp**(*other*)

For details, please refer to *mindspore.ops.ldexp()*.

mindspore.Tensor.le

Tensor.**le**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.le()*.

mindspore.Tensor.lerp

Tensor.**lerp**(*end*, *weight*) → *Tensor*
For more details, please refer to *mindspore.ops.lerp()*.

mindspore.Tensor.less

Tensor.**less**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.less()*.

mindspore.Tensor.less_equal

Tensor.**less_equal**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.less_equal()*.

mindspore.Tensor.log

Tensor.**log**() → *Tensor*
For details, please refer to *mindspore.ops.log()*.

mindspore.Tensor.log10

Tensor.**log10**() → *Tensor*
For details, please refer to *mindspore.ops.log10()*.

mindspore.Tensor.log1p

Tensor.**log1p**() → *Tensor*
For details, please refer to *mindspore.ops.log1p()*.

mindspore.Tensor.log2

Tensor.**log2**() → *Tensor*
For details, please refer to *mindspore.ops.log2()*.

mindspore.Tensor.logaddexp

Tensor.**logaddexp**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.logaddexp()*.

mindspore.Tensor.logaddexp2

Tensor.**logaddexp2**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.logaddexp2()*.

mindspore.Tensor.logcumsumexp

Tensor.**logcumsumexp**(*axis*)
For details, please refer to *mindspore.ops.logcumsumexp()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.logdet

Tensor.**logdet**()
For details, please refer to *mindspore.ops.logdet()*.

mindspore.Tensor.logical_and

Tensor.**logical_and**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.logical_and()*.

mindspore.Tensor.logical_not

Tensor.**logical_not**() → *Tensor*
For details, please refer to *mindspore.ops.logical_not()*.

mindspore.Tensor.logical_or

Tensor.**logical_or**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.logical_or()*.

mindspore.Tensor.logical_xor

Tensor.**logical_xor**(*other*) → *Tensor*
Computes the "logical XOR" of two tensors element-wise.

$$out_i = self_i \oplus other_i$$

Note:

- *self* and *other* comply with the type conversion rules to make the data types consistent.
- When the *other* is bool, it could only be a constant.

Parameters

other (*Union[Tensor, bool]*) – A bool or a tensor whose data type can be implicitly converted to bool.

Returns

Tensor, the shape is the same as the *self* and *other* after broadcasting, and the data type is bool.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([True, False, True]), mindspore.bool_)
>>> other = Tensor(np.array([True, True, False]), mindspore.bool_)
>>> output = input.logical_xor(other)
>>> print(output)
[ False True True]
>>> x = Tensor(1, mindspore.bool_)
>>> other = Tensor(0, mindspore.bool_)
>>> output = input.logical_xor(other)
>>> print(output)
True
```

mindspore.Tensor.logit

`Tensor.`**`logit`**(*eps=None*)

For details, please refer to *mindspore.ops.logit()*.

mindspore.Tensor.logsumexp

`Tensor.`**`logsumexp`**(*dim*, *keepdim=False*) → *Tensor*

For details, please refer to *mindspore.ops.logsumexp()*.

mindspore.Tensor.log_normal

`Tensor.`**`log_normal`**(*mean=1.0*, *std=2.0*)

Fills the elements of the input tensor with log normal values initialized by given mean and std:

$$f(x; 1.0, 2.0) = \frac{1}{x\delta\sqrt{2\pi}} e^{-\frac{(\ln x-\mu)^2}{2\delta^2}}$$

where μ, δ is mean and standard deviation of lognormal distribution respectively.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **mean** (*float, optional*) – the mean of normal distribution. With float data type. Default: 1.0.
- **std** (*float, optional*) – the std of normal distribution. With float data type. Default: 2.0.

Returns

Tensor. A Tensor with the same type and shape of input.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> x = mindspore.Tensor(np.array([[1, 2], [3, 4]]), dtype=mindspore.float32)
>>> output = x.log_normal()
>>> print(output)
[[1.2788825 2.3305743]
[14.944194 0.16303174]]
```

mindspore.Tensor.long

`Tensor.long()`

Converts input tensor dtype to *int64*. If the value in tensor is float or half, the decimal will be discarded.

Returns

Tensor, converted to the *int64* dtype.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor(np.ones([2,2]), mindspore.int32)
>>> output = input_x.long()
>>> print(output.dtype)
Int64
```

mindspore.Tensor.lt

Tensor.**lt**(*other*) → *Tensor*

For more details, please refer to *mindspore.Tensor.less()*.

mindspore.Tensor.lu_solve

Tensor.**lu_solve**(*LU_data*, *LU_pivots*)

For details, please refer to *mindspore.ops.lu_solve()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.masked_fill

Tensor.**masked_fill**(*mask*, *value*) → *Tensor*

For details, please refer to *mindspore.ops.masked_fill()*.

mindspore.Tensor.masked_fill_

Tensor.**masked_fill_**(*mask*, *value*) → *Tensor*

In-place version of *mindspore.Tensor.masked_fill()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.masked_scatter

Tensor.**masked_scatter**(*mask*, *x*)

Updates the value in the "self Tensor" with the *tensor* value according to the mask, and returns a Tensor. The shape of *mask* and the "self Tensor" must be the same or *mask* is broadcastable.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **mask** (Tensor[bool]) −A bool tensor with a shape broadcastable to the "self Tensor".
- **x** (Tensor) −A tensor with the same data type as the "self Tensor". The number of elements must be greater than or equal to the number of True's in *mask*.

Returns

Tensor, with the same type and shape as the "self Tensor".

Raises

- **TypeError** −If *mask* or *x* is not a Tensor.
- **TypeError** −If data type of the "self Tensor" is not be supported.

- **TypeError** –If dtype of *mask* is not bool.
- **TypeError** –If the dim of the "self Tensor" less than the dim of *mask*.
- **ValueError** –If *mask* can not be broadcastable to the "self Tensor".
- **ValueError** –If the number of elements in *x* is less than the number required for the updates.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([1., 2., 3., 4.]), mindspore.float32)
>>> mask = Tensor(np.array([True, True, False, True]), mindspore.bool_)
>>> tensor = Tensor(np.array([5., 6., 7.]), mindspore.float32)
>>> output = x.masked_scatter(mask, tensor)
>>> print(output)
[5. 6. 3. 7.]
```

mindspore.Tensor.masked_select

Tensor.**masked_select**(*mask*) → *Tensor*

For details, please refer to *mindspore.ops.masked_select()*.

mindspore.Tensor.matmul

Tensor.**matmul**(*tensor2*) → Union[*Tensor*, numbers.Number]

Returns the matrix product of two tensors.

Note:

- Numpy arguments *out*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.
- The dtype of *self* and *tensor2* must be same.
- On Ascend platform, the dims of *self* and *tensor2* must be between 1 and 6.
- On GPU platform, the supported dtypes of *self* and *tensor2* are ms.float16 and ms.float32.

Parameters

tensor2 (Tensor) –Input tensor, scalar not allowed. The last dimension of *self* must be the same size as the second last dimension of *tensor2*. And the shape of tensor and other could be broadcast.

Returns

Tensor or scalar, the matrix product of the inputs. This is a scalar only when both *self* and *tensor2* are 1-d vectors.

Raises

- **TypeError** –If the dtype of *self* and the dtype of *tensor2* are not the same.

- **ValueError** –If the last dimension of *self* is not the same size as the second-to-last dimension of *tensor2*, or if a scalar value is passed in.
- **ValueError** –If the shape of *self* and *tensor2* could not broadcast together.
- **RuntimeError** –On Ascend platforms, the dims of *self* or *tensor2* is less than 1 or greater than 6.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> # case 1 : Reasonable application of broadcast mechanism
>>> input = Tensor(np.arange(2 * 3 * 4).reshape(2, 3, 4), mindspore.float32)
>>> other = Tensor(np.arange(4 * 5).reshape(4, 5), mindspore.float32)
>>> output = input.matmul(other)
>>> print(output)
[[[  70.   76.   82.   88.   94.]
  [ 190.  212.  234.  256.  278.]
  [ 310.  348.  386.  424.  462.]]
 [[ 430.  484.  538.  592.  646.]
  [ 550.  620.  690.  760.  830.]
  [ 670.  756.  842.  928. 1014.]]]
>>> print(output.shape)
(2, 3, 5)
>>> # case 2 : the rank of `tensor2` is 1
>>> input = Tensor(np.ones([1, 2]), mindspore.float32)
>>> other = Tensor(np.ones([2,]), mindspore.float32)
>>> output = input.matmul(other)
>>> print(output)
[2.]
>>> print(output.shape)
(1,)
```

mindspore.Tensor.max

Tensor.**max**() → *Tensor*

Returns the maximum value of the self tensor.

Returns

Scalar Tensor with the same dtype as *input*, the maximum value of the input.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output = x.max()
>>> print(output)
[0.7]
```

`Tensor.max`(*dim*, *keepdim=False*)

Calculates the maximum value along with the given dim for the input tensor, and returns the maximum values and indices.

Parameters

- **dim** (*int*) –The dimension to reduce.
- **keepdim** (*bool, optional*) –Whether to keep dimension, if `True` the output will keep the same dimension as the *self* , the output will reduce dimension if `False`. Default: `False`.

Returns

tuple (Tensor), tuple of 2 tensors, containing the maximum value of the self tensor along the given dimension *dim* and the corresponding index.

- **values** (Tensor) - The maximum value of self tensor, with the same shape as *index*, and same dtype as *self*.
- **index** (Tensor) - The index for the maximum value of the self tensor, with dtype int64. If *keepdim* is `True` , the shape of output tensors is $(self_1, self_2, ..., self_{dim-1}, 1, self_{dim+1}, ..., self_N)$. Otherwise, the shape is $(self_1, self_2, ..., self_{dim-1}, self_{dim+1}, ..., self_N)$.

Raises

- **TypeError** –If *keepdim* is not a bool.
- **TypeError** –If *dim* is not an int.
- **TypeError** –If self tensor data type is Complex.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output, index = x.max(0, keepdim=True)
>>> print(output, index)
[0.7] [3]
```

`Tensor.max`(*axis=None*, *keepdims=False*, *, *initial=None*, *where=True*, *return_indices=False*)

Return the maximum of a tensor or maximum along an axis.

Note: When *axis* is `None`, *keepdims* and subsequent parameters have no effect. At the same time, the index is fixed to return 0.

Parameters

- **axis** (*Union[None, int, list, tuple of ints], optional*) – Axis or axes along which to operate. By default, flattened input is used. If this is a tuple of ints, the maximum is selected over multiple axes, instead of a single axis or all the axes as before. Default: `None` .
- **keepdims** (*bool, optional*) – If this is set to `True` , the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. Default: `False` .

Keyword Arguments

- **initial** (*scalar, optional*) – The minimum value of an output element. Must be present to allow computation on empty slice. Default: `None` .
- **where** (*bool Tensor, optional*) – A boolean tensor which is broadcasted to match the dimensions of array, and selects elements to include in the reduction. If non-default value is passed, initial must also be provided. Default: `True` .
- **return_indices** (*bool, optional*) – Whether to return the index of the maximum value. Default: `False` . If *axis* is a list or tuple of ints, it must be `False` .

Returns

Tensor or scalar, maximum of self tensor. If *axis* is `None` , the result is a scalar value. If *axis* is given, the result is a tensor of dimension `self.ndim - 1`.

Raises

TypeError – If arguments have types not specified above.

See also:

- *mindspore.Tensor.argmin()*: Return the indices of the minimum values along an axis.
- *mindspore.Tensor.argmax()*: Return the indices of the maximum values along an axis.
- *mindspore.Tensor.min()*: Return the minimum of a tensor or minimum along an axis.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> a = Tensor(np.arange(4).reshape((2, 2)).astype('float32'))
>>> output = a.max()
>>> print(output)
3.0
>>> value, indices = a.max(axis=0, return_indices=True)
>>> print(value)
[2. 3.]
>>> print(indices)
[1 1]
```

mindspore.Tensor.maximum

Tensor.**maximum**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.maximum()*.

mindspore.Tensor.mean

Tensor.**mean**(*dim=None*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Reduces all dimension of a tensor by averaging all elements in the dimension, by default. And reduce a dimension of *self* along the specified *dim*. *keepdim* determines whether the dimensions of the output and self are the same.

Note: The *dim* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **dim** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. Default: None , reduce all dimensions. Only constant value is allowed. Assume the rank of *self* is r, and the value range is [-r,r).
- **keepdim** (*bool, optional*) –If True , keep these reduced dimensions and the length is 1. If False , don't keep these dimensions. Default: False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: None .

Returns

Tensor, has the same data type as self tensor.

- If *dim* is None , and *keepdim* is False , the output is a 0-D tensor representing the product of all elements in the self tensor.
- If *dim* is int, set as 1, and *keepdim* is False , the shape of output is $(x_0, x_2, ..., x_R)$.
- If *dim* is tuple(int), set as (1, 2), and *keepdim* is False , the shape of output is $(x_0, x_3, ..., x_R)$.
- If *dim* is 1-D Tensor, set as [1, 2], and *keepdim* is False , the shape of output is $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *dim* is not one of the following: int, tuple, list or Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = Tensor.mean(x, 1, keepdim=True)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by averaging all elements in the dimension.
>>> x = Tensor(np.array([[[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[6, 6, 6, 6, 6, 6], [8, 8, 8, 8, 8, 8], [10, 10, 10, 10, 10, 10]]]),
... mindspore.float32)
>>> output = Tensor.mean(x)
>>> print(output)
5.0
>>> print(output.shape)
()
>>> # case 2: Reduces a dimension along the dim 0
>>> output = Tensor.mean(x, 0, True)
>>> print(output)
[[[4. 4. 4. 4. 4. 4.]
  [5. 5. 5. 5. 5. 5.]
  [6. 6. 6. 6. 6. 6.]]]
>>> # case 3: Reduces a dimension along the dim 1
>>> output = Tensor.mean(x, 1, True)
>>> print(output)
[[[2. 2. 2. 2. 2. 2.]]
 [[5. 5. 5. 5. 5. 5.]]
 [[8. 8. 8. 8. 8. 8.]]]
>>> # case 4: Reduces a dimension along the dim 2
>>> output = Tensor.mean(x, 2, True)
>>> print(output)
[[[ 2.]
  [ 2.]
  [ 2.]]
 [[ 4.]
  [ 5.]
  [ 6.]]
 [[ 6.]
  [ 8.]
  [10.]]]
```

`Tensor.`**`mean`**(*axis=None*, *keep_dims=False*) → *Tensor*

For details, please refer to *`mindspore.ops.mean()`* .

mindspore.Tensor.median

Tensor.**median**(*axis=- 1*, *keepdims=False*) → Tuple of Tensors

Computes the median and indices of input tensor.

Warning:

- *indices* does not necessarily contain the first occurrence of each median value found in the *input*, unless it is unique. The specific implementation of this API is device-specific. The results may be different on CPU and GPU.

Parameters

- **axis** (*int, optional*) –Specify the axis for calculation. Default: `-1` .
- **keepdims** (*bool, optional*) –Whether the output tensor need to retain *axis* dimension or not. Default: `False` .

Returns

- y (Tensor) - Returns the median value along the specified dimension. And It has the same dtype as the *input*.
- indices (Tensor) - The index of the median. And the dtype is int64.

Raises

- **TypeError** –If *axis* is not an int.
- **TypeError** –If *keepdims* is not a bool.
- **ValueError** –If *axis* is not in range of [-x.dim, x.dim-1].

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[0.57, 0.11, 0.21],[0.38, 0.50, 0.57], [0.36, 0.16, 0.44]]).
↪astype(np.float32))
>>> y = x.median(axis=0, keepdims=False)
>>> print(y)
(Tensor(shape=[3], dtype=Float32, value= [ 3.79999995e-01,  1.59999996e-01,  4.39999998e-
↪01]),
Tensor(shape=[3], dtype=Int64, value= [1, 2, 2]))
```

Tensor.**median**() → *Tensor*

Return the median of the input.

Returns

- y (Tensor) - Output median.

Supported Platforms:

Ascend

Tensor.**median**(*dim=- 1*, *keepdim=False*) → Tuple of Tensors

Output the median on the specified dimension `dim` and its corresponding index. If `dim` is None, calculate the median of all elements in the Tensor.

Parameters

- **dim** (*int, optional*) –Specify the axis for calculation. Default: `None` .
- **keepdim** (*bool, optional*) –Whether the output tensor need to retain `dim` dimension or not. Default: `False` .

Returns

- y (Tensor) - Output median, with the same data type as `input` .
  - If `dim` is `None` , `y` only has one element.
  - If `keepdim` is `True` , the `y` has the same shape as the `input` except the shape of `y` in dimension *dim* is size 1.
  - Otherwise, the `y` lacks *dim* dimension than input.
- indices (Tensor) - The index of the median. Shape is consistent with `y` , with a data type of int64.

Raises

- **TypeError** –If `dim` is not an int.
- **TypeError** –If `keepdim` is not a bool.
- **ValueError** –If `dim` is not in range of [-x.dim, x.dim-1].

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[0.57, 0.11, 0.21],[0.38, 0.50, 0.57], [0.36, 0.16, 0.44]]).
↪astype(np.float32))
>>> y = x.median(dim=0, keepdim=False)
>>> print(y)
(Tensor(shape=[3], dtype=Float32, value= [ 3.79999995e-01,  1.59999996e-01,  4.39999998e-
↪01]),
Tensor(shape=[3], dtype=Int64, value= [1, 2, 2]))
```

mindspore.Tensor.t

`Tensor.t()`

Transpose *self* .

For details, please refer to *mindspore.ops.t()*.

Supported Platforms:

`Ascend`

mindspore.Tensor.mH

property Tensor.**mH**

Accessing this property is equivalent to Calling self.adjoint(). For details, please refer to *mindspore.ops.adjoint()*.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[0. + 0.j, 1. + 1.j], [2. + 2.j, 3. + 3.j]]))
>>> output = x.mH
>>> print(output)
[[0.-0.j 2.-2.j]
 [1.-1.j 3.-3.j]]
```

mindspore.Tensor.min

Tensor.**min**() → *Tensor*

Returns the minimum value of the self tensor.

Returns

Scalar Tensor with the same dtype as *self*, the minimum value of the self.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output = Tensor.min(x)
>>> print(output)
0.0
```

Tensor.**min**(*dim*, *keepdim=False*)

Calculates the minimum value along with the given dim for the self tensor, and returns the minimum values and indices.

Parameters

- **dim** (*int*) −The dimension to reduce.
- **keepdim** (*bool, optional*) −Whether to reduce dimension, if True the output will keep the same dimension as the *self* , the output will reduce dimension if False. Default: False.

Returns

tuple (Tensor), tuple of 2 tensors, containing the minimum value of the self tensor along the given dimension *dim* and the corresponding index.

- **values** (Tensor) - The minimum value of self tensor along the given dimension *dim*, with the same shape as *index*, and same dtype as *self*.
- **index** (Tensor) - The index for the minimum value of the self tensor, with dtype int64. If *keepdim* is `True`, the shape of output tensors is $(self_1, self_2, ..., self_{dim-1}, 1, self_{dim+1}, ..., self_N)$. Otherwise, the shape is $(self_1, self_2, ..., self_{dim-1}, self_{dim+1}, ..., self_N)$.

Raises

- **TypeError** – If *keepdim* is not a bool.
- **TypeError** – If *dim* is not an int.
- **TypeError** – If self tensor data type is Complex.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output, index = x.min(0, keepdim=True)
>>> print(output, index)
[0.0] [0]
```

`Tensor.`**`min`**(*axis=None*, *keepdims=False*, *, *initial=None*, *where=True*, *return_indices=False*) → *Tensor* | number.Number

Return the minimum of a tensor or minimum along an axis.

Note: When *axis* is `None`, *keepdims* and subsequent parameters have no effect. At the same time, the index is fixed to return 0.

Parameters

- **axis** (*Union[None, int, list, tuple of ints], optional*) – An axis or axes along which to operate. By default, flattened input is used. If *axis* is a tuple of ints, the minimum is selected over multiple axes, instead of a single axis or all the axes as before. Default: `None`.
- **keepdims** (*bool, optional*) – If `True`, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. Default: `False`.

Keyword Arguments

- **initial** (*scalar, optional*) – The minimum value of an output element. Must be present to allow computation on empty slice. Default: `None`.
- **where** (*Tensor[bool], optional*) – A boolean tensor which is broadcasted to match the dimensions of array, and selects elements to include in the reduction. If non-default value is passed, initial must also be provided. Default: `True`.
- **return_indices** (*bool, optional*) – Whether to return the index of the minimum value. Default: `False`. If *axis* is a list or tuple of ints, it must be `False`.

Returns

Tensor or scalar, minimum of self tensor. If *axis* is `None` , the result is a scalar value. If *axis* is given, the result is a tensor of dimension `self.ndim - 1`.

Raises

`TypeError` –If arguments have types not specified above.

See also:

- *`mindspore.Tensor.argmin()`*: Return the indices of the minimum values along an axis.
- *`mindspore.Tensor.argmax()`*: Return the indices of the maximum values along an axis.
- *`mindspore.Tensor.max()`*: Return the minimum of a tensor or minimum along an axis.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> a = Tensor(np.arange(4).reshape((2, 2)).astype('float32'))
>>> output = Tensor.min(a)
>>> print(output)
0.0
>>> output = Tensor.min(a, axis=0)
>>> print(output)
[0. 1.]
>>> output = Tensor.min(a, axis=0, initial=9, where=Tensor([False]))
>>> print(output)
[9. 9.]
>>> output = Tensor.min(a, axis=0, initial=9, where=Tensor([False, True]))
>>> print(output)
[9. 1.]
>>> value, indices = Tensor.min(a, axis=0, return_indices=True)
>>> print(value)
[0. 1.]
>>> print(indices)
[0 0]
```

mindspore.Tensor.minimum

`Tensor.`**`minimum`**(*other*) → *Tensor*

For details, please refer to *`mindspore.ops.minimum()`*.

mindspore.Tensor.mm

Tensor.**mm**(*mat2*) → *Tensor*

Returns the matrix product of two arrays. If *self* is a $(n \times m)$ Tensor, *mat2* is a $(m \times p)$ Tensor, *out* will be a $(n \times p)$ Tensor.

Note: This function cannot support broadcasting. Refer to *mindspore.ops.matmul()* instead if you need a broadcastable function.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

mat2 (Tensor) −The second matrix of matrix multiplication. The last dimension of *self* must be the same size as the first dimension of *mat2*.

Returns

Tensor, the matrix product of the inputs.

Raises

- **TypeError** −If *self* or *mat2* is not a Tensor.
- **RuntimeError** −If the last dimension of *self* is not the same size as the second-to-last dimension of *mat2*.
- **RuntimeError** −If dtype of *self* or *mat2* is not float16, float32 or bfloat16.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x1 = ms.Tensor(np.random.rand(2, 3), ms.float32)
>>> x2 = ms.Tensor(np.random.rand(3, 4), ms.float32)
>>> out = x1.mm(x2)
>>> print(out.shape)
(2, 4)
```

mindspore.Tensor.moveaxis

Tensor.**moveaxis**(*source*, *destination*)

For details, please refer to *mindspore.ops.moveaxis()*.

mindspore.Tensor.movedim

Tensor.**movedim**(*source*, *destination*)
For details, please refer to *mindspore.ops.movedim()*.

mindspore.Tensor.move_to

Tensor.**move_to**(*to*, *blocking=True*)
Copy Tensor to target device synchronously or asynchronously, default synchronously. only support PyNative mode.

Parameters

- **to** (*str*) –a string type value, one of "Ascend", "GPU", "CPU".
- **blocking** (*bool*) –a bool type value, using synchronous copy or asynchronous copy. Default: True , synchronous copy.

Returns

New Tensor, storged on target device which with the same type and shape as the "self Tensor".

Raises

- **ValueError** –If the type of *blocking* is not bool type.
- **ValueError** –If the value of *to* is not one of "Ascend", "GPU", "CPU".
- **ValueError** –If the run mode is not PyNative mode.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> x = ms.Tensor([1, 2, 3], ms.int64)
>>> new_tensor = x.move_to("CPU")
```

mindspore.Tensor.msort

Tensor.**msort**()
For details, please refer to *mindspore.ops.msort()*.

mindspore.Tensor.mT

property Tensor.**mT**

Returns the Tensor that exchanges the last two dimensions. Accessing the attribute, x.mT, is equal to calling the method, x.swapaxes(-2, -1). For details, please refer to *mindspore.Tensor.swapaxes()*.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.ones((2, 3, 4)))
>>> output = x.mT
>>> print(output.shape)
(2, 4, 3)
```

mindspore.Tensor.mul

Tensor.**mul**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.mul()*.

mindspore.Tensor.mul_

Tensor.**mul_**(*other*) → *Tensor*

Multiplies two tensors element-wise.

$$out_i = tensor_i * other_i$$

Warning: This is an experimental API that is subject to change or deletion.

Note:

- When *self* and *other* have different shapes, *other* be able to broadcast to a *self*.
- *self* and *other* can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.

Parameters

other (*Union[Tensor, number.Number, bool]*) – *other* is a number.Number or a bool or a tensor whose data type is number.Number and bool.

Returns

Tensor, the shape is the same as *self* , and the data type is the same as *self* .

Raises

- **TypeError** – If *other* is not one of the following: Tensor, number.Number, bool.
- **RuntimeError** – If *other* cannot be broadcast to *self*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([4.0, 5.0, 6.0]), mindspore.float32)
>>> output = x.mul_(y)
>>> print(output)
[ 4. 10. 18.]
>>> print(x)
[ 4. 10. 18.]
```

mindspore.Tensor.multinomial

Tensor.**multinomial**(*num_samples*, *replacement=True*, *seed=None*)

For details, please refer to *mindspore.ops.multinomial()*.

mindspore.Tensor.multiply

Tensor.**multiply**(*value*)

For details, please refer to *mindspore.ops.mul()*. The parameter *value* of the current interface is the same as the parameter *other* of the reference interface.

mindspore.Tensor.mvlgamma

Tensor.**mvlgamma**(*p*)

For details, please refer to *mindspore.ops.mvlgamma()*.

mindspore.Tensor.nan_to_num

Tensor.**nan_to_num**(*nan=None*, *posinf=None*, *neginf=None*) → *Tensor*

For details, please refer to *mindspore.ops.nan_to_num()*.

Supported Platforms:

Ascend CPU

mindspore.Tensor.nanmean

Tensor.**nanmean**(*axis=None*, *keepdims=False*, *, *dtype=None*)
For details, please refer to *mindspore.ops.nanmean()*.

mindspore.Tensor.nanmedian

Tensor.**nanmedian**(*axis=- 1*, *keepdims=False*)
For details, please refer to *mindspore.ops.nanmedian()*.

mindspore.Tensor.nansum

Tensor.**nansum**(*dim=None*, *keepdim=False*, *, *dtype=None*) → *Tensor*
Computes sum of input Tensor over a given dimension, treating NaNs as zero.

Warning:

- It is only supported on Atlas A2 Training Series Products.
- This is an experimental API that is subject to change or deletion.

Parameters

- **dim** (*Union[int, tuple(int)], optional*) –The dimensions to sum. Dim must be in the range [-rank(self), rank(self)). Default: None, which indicates the sum of all elements in a tensor.
- **keepdim** (*bool, optional*) –Whether the output Tensor keeps dimensions or not. Default: False, indicating that no dimension is kept.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The dtype of output Tensor. Default: None.

Returns

Tensor, the sum of input Tensor in the given dimension dim, treating NaNs as zero.

- If dim is None, keepdim is False, the output is a 0-D Tensor representing the sum of all elements in the self Tensor.
- If dim is int, set as 2, and keepdim is False, the shape of output is $(self_1, self_3, ..., self_R)$.
- If dim is tuple(int) or list(int), set as (2, 3), and keepdim is False, the shape of output is $(self_1, self_4, ..., self_R)$.

Raises

- **TypeError** –If *keepdim* is not a bool.
- **TypeError** –If the dtype of *self* or *dtype* is complex type.
- **ValueError** –If *dim* is not in [-rank(self), rank(self)).

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[float("nan"), 2, 3], [1, 2, float("nan")]]), mindspore.float32)
>>> output1 = x.nansum(dim=0, keepdim=False, dtype=mindspore.float32)
>>> output2 = x.nansum(dim=0, keepdim=True, dtype=mindspore.float32)
>>> print(output1)
[1. 4. 3.]
>>> print(output2)
[[1. 4. 3.]]
```

Tensor.**nansum**(*axis=None*, *keepdims=False*, *, *dtype=None*) → *Tensor*

Computes sum of *input* over a given dimension, treating NaNs as zero.

Parameters

- **axis** (*Union[int, tuple(int)], optional*) – The dimensions to reduce. Supposed the rank of *self* is r, axis must be in the range [-r,r). Default: None, all dimensions are reduced.
- **keepdims** (*bool, optional*) – Whether the output Tensor keeps dimensions or not. Default: False.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The dtype of output Tensor. Default: None.

Returns

Tensor, the sum of input Tensor in the given dimension dim, treating NaNs as zero.

- If axis is None, keepdims is False, the output is a 0-D Tensor representing the sum of all elements in the input Tensor.
- If axis is int, set as 2, and keepdims is False, the shape of output is $(self_1, self_3, ..., self_R)$.
- If axis is tuple(int) or list(int), set as (2, 3), and keepdims is False, the shape of output is $(self_1, self_4, ..., self_R)$.

Raises

- **TypeError** – If *keepdims* is not a bool.
- **TypeError** – If the dtype of *self* or *dtype* is complex type.
- **ValueError** – If *axis* not in [-rank(self), rank(self)).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[float("nan"), 2, 3], [1, 2, float("nan")]]), mindspore.float32)
>>> output1 = x.nansum(axis=0, keepdims=False, dtype=mindspore.float32)
>>> output2 = x.nansum(axis=0, keepdims=True, dtype=mindspore.float32)
>>> print(output1)
[1. 4. 3.]
```

(continues on next page)

(continued from previous page)

```
>>> print(output2)
[[1. 4. 3.]]
```

mindspore.Tensor.narrow

Tensor.**narrow**(*dim*, *start*, *length*) → *Tensor*

Obtains a tensor of a specified length at a specified start position along a specified axis.

Parameters

- **dim** (*int*) – the axis along which to narrow.
- **start** (*Union[int, Tensor]*) – the starting dimension.
- **length** (*int*) – the distance to the ending dimension.

Returns

output (Tensors) - The narrowed tensor.

Raises

- **ValueError** – The value of *dim* is out of range [-self.ndim, self.ndim).
- **ValueError** – The value of *start* is out of range [-self.shape[dim], self.shape[dim]].
- **ValueError** – The value of *length* is out of range [0, self.shape[dim] - start].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.int32)
>>> output = x.narrow(0, 0, 2)
>>> print(output)
[[ 1 2 3]
 [ 4 5 6]]
>>> output = x.narrow(1, 1, 2)
>>> print(output)
[[ 2 3]
 [ 5 6]
 [ 8 9]]
```

mindspore.Tensor.nbytes

property Tensor.**nbytes**

Return the total number of bytes taken by the tensor.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.nbytes
>>> print(output)
32
```

mindspore.Tensor.ndim

property Tensor.**ndim**

Return the number of tensor dimensions.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.ndim
>>> print(output)
2
```

mindspore.Tensor.ndimension

Tensor.**ndimension**()

Alias for *mindspore.Tensor.ndim*.

mindspore.Tensor.ne

Tensor.**ne**(*other*) → *Tensor*

Alias for *mindspore.Tensor.not_equal()*.

mindspore.Tensor.neg

Tensor.**neg**() → *Tensor*

For details, please refer to *mindspore.ops.neg()*.

mindspore.Tensor.negative

Tensor.**negative**() → *Tensor*
Alias for *mindspore.Tensor.neg()*.

mindspore.Tensor.nelement

Tensor.**nelement**()
Alias for *mindspore.Tensor.numel()*.

mindspore.Tensor.new_ones

Tensor.**new_ones**(*size*, *dtype=None*) → *Tensor*
Return a tensor of *size* filled with ones.

Parameters

- **size** (*Union[int, tuple(int), list(int)]*) – An int, list or tuple of integers defining the output shape.
- **dtype** (*mindspore.dtype*, optional) – The desired dtype of the output tensor. If None, the returned tensor has the same dtype as *self*. Default: None.

Returns

Tensor, the shape and dtype is defined above and filled with ones.

Raises

- **TypeError** – If *size* is neither an int nor a tuple/list of int.
- **TypeError** – If *dtype* is not a MindSpore dtype.
- **ValueError** – If *size* contains negative values.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor((), mindspore.int32)
>>> x.new_ones((2, 3))
Tensor(shape=[2, 3], dtype=Int32, value=
[[1, 1, 1],
 [1, 1, 1]])
```

mindspore.Tensor.new_zeros

Tensor.**new_zeros**(*size*, *dtype=None*) → *Tensor*

Return a tensor of *size* filled with zeros.

Parameters

- **size** (*Union[int, tuple(int), list(int)]*) – An int, list or tuple of integers defining the output shape.
- **dtype** (*mindspore.dtype*, optional) – The desired dtype of the output tensor. If None, the returned tensor has the same dtype as *self*. Default: None.

Returns

Tensor, the shape and dtype is defined above and filled with zeros.

Raises

- **TypeError** – If *size* is neither an int nor a tuple/list of int.
- **TypeError** – If *dtype* is not a MindSpore dtype.
- **ValueError** – If *size* contains negative values.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor((), mindspore.int32)
>>> x.new_zeros((2, 3))
Tensor(shape=[2, 3], dtype=Int32, value=
[[0, 0, 0],
 [0, 0, 0]])
```

mindspore.Tensor.nextafter

Tensor.**nextafter**(*other*)

For details, please refer to *mindspore.ops.nextafter()*.

mindspore.Tensor.nonzero

Tensor.**nonzero**(*, *as_tuple=False*)

Return the positions of all non-zero values.

Note: The rank of *self*.

- Ascend: its rank can be equal to 0 except O2 mode.
- CPU/GPU: its rank should be greater than or eaqual to 1.

Keyword Arguments

as_tuple (bool, *optional*) –Whether the output is tuple. If `False` , return Tensor. Default: `False` . If `True` , return Tuple of Tensor, only support `Ascend` .

Returns

- If *as_tuple* is `False`, return the Tensor, a 2-D Tensor whose data type is int64, containing the positions of all non-zero values of the *self* .
- If *as_tuple* is `True`, return the Tuple of Tensor and data type is int64. The Tuple length is the dimension of the *self* tensor, and each element is the 1D tensor of the subscript of all non-zero elements of the *self* tensor in that dimension.

Raises

- **TypeError** –If *self* is not Tensor.
- **TypeError** –If *as_tuple* is not bool.
- **RuntimeError** –On GPU or CPU or Ascend O2 mode, if dim of *input* equals to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[[1,  0], [-5, 0]]]), mindspore.int32)
>>> output = x.nonzero()
>>> print(output)
[[0 0 0]
[0 1 0]]
>>> x = Tensor(np.array([1, 0, 2, 0, 3]), mindspore.int32)
>>> output = x.nonzero(as_tuple=False)
>>> print(output)
[[0]
[2]
[4]]
>>> x = Tensor(np.array([[[1,  0], [-5, 0]]]), mindspore.int32)
>>> output = x.nonzero(as_tuple=True)
>>> print(output)
(Tensor(shape=[2], dtype=Int64, value=[0, 0]),
Tensor(shape=[2], dtype=Int64, value=[0, 1]),
Tensor(shape=[2], dtype=Int64, value=[0, 0]))
>>> x = Tensor(np.array([1, 0, 2, 0, 3]), mindspore.int32)
>>> output = x.nonzero(as_tuple=True)
>>> print(output)
(Tensor(shape=[3], dtype=Int64, value=[0, 2, 4]), )
```

mindspore.Tensor.norm

Tensor.**norm**(*ord=None*, *dim=None*, *keepdim=False*, *, *dtype=None*)
For details, please refer to *mindspore.ops.norm()*.

mindspore.Tensor.normal_

Tensor.**normal_**(*mean=0*, *std=1*, *, *generator=None*)
Update the *self* tensor in place by generating random numbers sampled from the normal distribution which constructed by the parameters *mean* and *std*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **mean** (*number, optional*) – the mean of normal distribution. With float data type. Default: 0.
- **std** (*number, optional*) – the std of normal distribution. With float data type. Default: 1.

Keyword Arguments

generator (*mindspore.Generator*, optional) – a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.

Returns

A tensor that is filled with random numbers that follow a normal distribution and that has the same type and shape as the *self* tensor.

Raises

TypeError – If the dtype of *mean* or *std* is not one of: bool, int, float, complex.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> x = mindspore.Tensor(np.array([[1, 2], [3, 4]]), dtype=mindspore.float32)
>>> output = x.normal_()
>>> print(output)
[[0.2788825 1.3305743]
 [1.244194 1.16303174]]
```

mindspore.Tensor.not_equal

Tensor.**not_equal**(*other*) → *Tensor*
For details, please refer to *mindspore.ops.ne()*.

mindspore.Tensor.numel

Tensor.**numel**()
For details, please refer to *mindspore.ops.numel()*.

mindspore.Tensor.numpy

Tensor.**numpy**()
Alias for *mindspore.Tensor.asnumpy()*.

mindspore.Tensor.orgqr

Tensor.**orgqr**(*input2*)
For details, please refer to *mindspore.ops.orgqr()*.

mindspore.Tensor.ormqr

Tensor.**ormqr**(*input2*, *input3*, *left=True*, *transpose=False*)
For details, please refer to *mindspore.ops.ormqr()*, Args *input2* and *input3* correspond to the args *tau* and *other* of *mindspore.ops.ormqr()*.

mindspore.Tensor.outer

Tensor.**outer**(*vec2*) → *Tensor*
For details, please refer to *mindspore.ops.outer()*.

mindspore.Tensor.permute

Tensor.**permute**(**axis*)
Tensor.permute supports unpacking the *axis* argument automatically when it is passed as an indefinite number of positional arguments, which has a slight difference from the input parameter of *mindspore.ops.permute()*. For details, please refer to *mindspore.ops.permute()*.

mindspore.Tensor.positive

Tensor.**positive**()
For details, please refer to *mindspore.ops.positive()*.

mindspore.Tensor.pow

Tensor.**pow**(*exponent*) → *Tensor*
For details, please refer to *mindspore.ops.pow()*.

mindspore.Tensor.prod

Tensor.**prod**(*dim=None*, *keepdim=False*, *dtype=None*) → *Tensor*
Reduces a dimension of a tensor by multiplying all elements in the dimension, by default. And also can reduce a dimension of *self* along the *dim*. Determine whether the dimensions of the output and self are the same by controlling *keepdim*.

Parameters

- **dim** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. Default: None , reduce all dimensions. Only constant value is allowed. Assume the rank of *self* is r, and the value range is [-r,r).
- **keepdim** (*bool, optional*) –If True , keep these reduced dimensions and the length is 1. If False , don't keep these dimensions. Default: False .
- **dtype** (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: None .

Returns

Tensor.

- If *dim* is None , and *keepdim* is False , the output is a 0-D tensor representing the product of all elements in the self tensor.
- If *dim* is int, set as 1, and *keepdim* is False , the shape of output is $(self_0, self_2, ..., self_R)$.
- If *dim* is tuple(int) or list(int), set as (1, 2), and *keepdim* is False , the shape of output is $(self_0, self_3, ..., self_R)$.
- If *dim* is 1-D Tensor, set as [1, 2], and *keepdim* is False , the shape of output is $(self_0, self_3, ..., self_R)$.

Raises

- **TypeError** –If *dim* is not one of the following: int, Tuple, list or Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = Tensor.prod(x, 1, True)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by multiplying all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = Tensor.prod(x)
>>> print(output)
2.2833798e+33
>>> print(output.shape)
()
>>> # case 2: Reduces a dimension along axis 0.
>>> output = Tensor.prod(x, 0, True)
>>> print(output)
[[[ 28.  28.  28.  28.  28.  28.]
  [ 80.  80.  80.  80.  80.  80.]
  [162. 162. 162. 162. 162. 162.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = Tensor.prod(x, 1, True)
>>> print(output)
[[[  6.   6.   6.   6.   6.   6.]]
 [[120. 120. 120. 120. 120. 120.]]
 [[504. 504. 504. 504. 504. 504.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = Tensor.prod(x, 2, True)
>>> print(output)
[[[1.00000e+00]
  [6.40000e+01]
  [7.29000e+02]]
 [[4.09600e+03]
  [1.56250e+04]
  [4.66560e+04]]
 [[1.17649e+05]
  [2.62144e+05]
  [5.31441e+05]]]
```

`Tensor.`**`prod`**(*axis=None*, *keep_dims=False*, *dtype=None*) → *Tensor*

For more details, please refer to *`mindspore.ops.prod()`*.

mindspore.Tensor.ptp

Tensor.**ptp**(*axis=None*, *keepdims=False*)

The name of the function comes from the acronym for "peak to peak". Calculate the difference between the maximum value and the minimum value along the axis.

Note: Numpy argument *out* is not supported.

Parameters

- **axis** (*Union[None, int, tuple(int)]*) – Axis or axes along which the range is computed. The default is to compute the variance of the flattened tensor. Default: `None` .
- **keepdims** (*bool*) – If this is set to `True` , the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the tensor. Default is `False` .

Returns

Tensor.

Raises

TypeError – If *self* is not a tensor, or *axis* and *keepdims* have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> x = Tensor([[4.0, 9.0, 2.0, 10.0], [6.0, 9.0, 7.0, 12.0]]).astype("float32")
>>> print(x.ptp(axis=1))
[8. 6.]
>>> print(x.ptp(axis=0))
[2. 0. 5. 2.]
```

mindspore.Tensor.rad2deg

Tensor.**rad2deg**()

For details, please refer to *mindspore.ops.rad2deg()*.

mindspore.Tensor.random_

Tensor.**random_**(*from_=0*, *to=None*, ***, *generator=None*)

Fill the tensor with numbers sampled from a discrete uniform distribution over an interval $[from\_, to - 1]$.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **from_** (*Union[number.Number, Tensor], optional*) –the lower bound of the generated random number. It can be a scalar value or a Tensor of any dimension with only a single element. Default: 0.
- **to** (*Union[number.Number, Tensor], optional*) –the upper bound of the generated random number. By default it's the upper limit of the input data type. It can be a scalar value or a Tensor of any dimension with only a single element. Default: None.

Keyword Arguments

generator (*mindspore.Generator*, optional) –a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.

Returns

The input tensor.

Raises

- **TypeError** –If *from_* or *to* is not integer.
- **RuntimeError** –If *from_* >= *to*.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor
>>> a = Tensor([[2, 3, 4], [1, 2, 3]])
>>> from_ = 0
>>> to = 5
>>> print(a.random_(from_, to).shape)
(2, 3)
```

mindspore.Tensor.random_categorical

Tensor.**random_categorical**(*num_sample*, *seed=0*, *dtype=mstype.int64*)

For details, please refer to *mindspore.ops.random_categorical()*.

mindspore.Tensor.ravel

Tensor.**ravel**()

Return a contiguous flattened tensor.

Returns

Tensor, a 1-D tensor, containing the same elements of the input.

See also:

- *mindspore.Tensor.reshape()*: Give a new shape to a tensor without changing its data.
- *mindspore.Tensor.flatten()*: Return a copy of the tensor collapsed into one dimension.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.ones((2,3,4), dtype=np.float32))
>>> output = x.ravel()
>>> print(output.shape)
(24,)
```

mindspore.Tensor.real

Tensor.**real**()
For details, please refer to *mindspore.ops.real()*.

mindspore.Tensor.reciprocal

Tensor.**reciprocal**() → *Tensor*
For details, please refer to *mindspore.ops.reciprocal()*.

mindspore.Tensor.register_hook

Tensor.**register_hook**(*hook*)
Registers a backward hook for tensor.

Note:

- The *hook* must be defined as the following code. *grad* is the gradient passed to the tensor, which may be modified by returning a new output gradient.
- The *hook* should have the following signature: hook(grad) -> New output gradient, but can not return None or not set return value.
- Higher-order differentiation does not support tensor *register_hook*.
- The following constraints must be met under graph mode:
  - The *hook* must satisfy the syntax constraints of the graph mode.
  - It is not supported to delete *hook* inside graph.
  - It is not supported to register *hook* after the *Tensor* is used before.
  - It is not supported to register multiple *hooks* for a *Tensor* inside graph.
  - Register *hook* in the graph will return then *Tensor* it self.

Parameters

hook (*function*) –Python function. Tensor backward hook function.

Returns

A handle corresponding to the *hook* . The handle can be used to remove the added *hook* by calling *handle.remove()* .

Raises

TypeError –If the *hook* is not a function of python.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def hook_fn(grad):
...     return grad * 2
...
>>> def hook_test(x, y):
...     z = x * y
...     z.register_hook(hook_fn)
...     z = z * y
...     return z
...
>>> ms_grad = ms.grad(hook_test, grad_position=(0,1))
>>> output = ms_grad(Tensor(1, ms.float32), Tensor(2, ms.float32))
>>> print(output)
(Tensor(shape=[], dtype=Float32, value=8), Tensor(shape=[], dtype=Float32, value=6))
```

mindspore.Tensor.remainder

Tensor.**remainder**(*other*) → *Tensor*

Computes the remainder of *self* divided by *other* element-wise. The result has the same sign as the divisor and its absolute value is less than that of *other*.

Supports broadcasting to a common shape and implicit type promotion.

```
remainder(input, other) == input - input.div(other, rounding_mode="floor") * other
```

Note: Complex inputs are not supported. At least one input need to be tensor, but not both are bool tensors.

The dividend *self* is a tensor whose data type is number or bool_.

Parameters

other (*Union[Tensor, numbers.Number, bool]*) –The divisor is a numbers.Number or a bool or a tensor whose data type is number or bool_ when the dividend is a tensor.

Returns

Tensor, with dtype promoted and shape broadcasted.

Raises

- **TypeError** –If *self* and *other* are not of types: (Tensor, Tensor), (Tensor, Number), (Tensor, bool), (Number, Tensor) or (bool, Tensor).
- **ValueError** –If *self* and *other* are not broadcastable.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]).astype(np.float32))
>>> y = Tensor(np.array([3.0, 2.0, 3.0]).astype(np.float64))
>>> output = x.remainder(y)
>>> print(output)
[2.  1.  0.]
```

Tensor.**remainder**(*divisor*) → *Tensor*

Computes the remainder of dividing the first input tensor by the second input tensor element-wise.

Inputs of *self* and *divisor* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, both dtypes cannot be bool, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

```
remainder(input, other) == input - input.div(other, rounding_mode="floor") * other
```

Warning:

- When the elements of input exceed 2048, there might be accuracy problems.
- The calculation results of this operator on Ascend and CPU might be inconsistent.
- If shape is expressed as (D1,D2⋯,Dn), then D1*D2⋯*DN<=1000000,n<=8.

Note: The first input *self* is a tensor whose data type is number.

Parameters

divisor (*Union[Tensor, numbers.Number, bool]*) – When the first input is a tensor, The second input could be a number, a bool or a tensor whose data type is number.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision.

Raises

- **TypeError** – If neither *self* nor *divisor* is one of the following: Tensor, Number, bool.
- **ValueError** – If the shape of *self* and *divisor* cannot be broadcasted to each other.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]).astype(np.float16))
>>> y = Tensor(np.array([3.0, 2.0, 3.0]).astype(np.float16))
>>> output = x.remainder(divisor=y)
>>> print(output)
[2.  1.  0.]
```

mindspore.Tensor.renorm

Tensor.**renorm**(*p*, *axis*, *maxnorm*)

For details, please refer to *mindspore.ops.renorm()*.

mindspore.Tensor.repeat

Tensor.**repeat**(**repeats*)

Copy the elements in each dimension of a Tensor based on the specified number of repetition times.

This function copies the tensor's data.

The shape of the output tensor can be described as follows, where n is the number of elements in *repeats*.

$$shape_i = \begin{cases} repeats_i * input.shape_i & \text{if } 0 \le i < input.rank \\ repeats_i & \text{if } input.rank \le i < n \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Note: If need to specify the number of repetition times for each element of a single dimension, please refer to *mindspore.Tensor.repeat_interleave()*.

Parameters

***repeats** (*int*) −Number of repetitions of *self* in each dimension. The value must be a non-negative number. `1` indicates that the dimension remains unchanged. The number of elements in *repeats* must be greater than or equals to the number of dimensions in *self* . When the number of dimensions of *self* is less than the number of elements of *repeats* , *self* is broadcasted to the number of dimensions with the same number of elements of *repeats* (as shown in the example).

Returns

Tensor, the new Tensor after the element is copied from the specified number of repetitions.

Raises

- **RuntimeError** −If the number of elements of *repeats* is less than the number of dimensions of *self* . Or *repeats* has negative element.
- **RuntimeError** −If the number of elements of *repeats* or the number of dimensions of *self* is larger than 8.
- **TypeError** −If type of *repeats* is unsupported.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor
>>> a = Tensor([[0, 1, 2], [3, 4, 5]])
>>> print(a.repeat(3, 2))
[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]]
>>> print(a.repeat(2, 1, 3))  # a is treated as a shape [1, 2, 3]
[[[0 1 2 0 1 2 0 1 2]
  [3 4 5 3 4 5 3 4 5]]
 [[0 1 2 0 1 2 0 1 2]
  [3 4 5 3 4 5 3 4 5]]]
```

`Tensor.`**`repeat`**(*repeats*) → *Tensor*

Copy the elements in each dimension of a Tensor based on the specified number of repetition times.

This function copies the tensor's data.

Expect that a variable-length int parameter is changed to a parameter which type is list or tuple, other operations are the same as the overload with **repeats* parameter.

The shape of the output tensor can be described as follows, where n is the number of elements in *repeats*.

$$shape_i = \begin{cases} repeats_i * input.shape_i & \text{if } 0 \leq i < input.rank \\ repeats_i & \text{if } input.rank \leq i < n \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Note: If need to specify the number of repetition times for each element of a single dimension, please refer to *mindspore.Tensor.repeat_interleave()*.

Parameters

repeats (*Union[tuple[int], list[int]]*) – Number of repetitions of *self* in each dimension. The value must be a non-negative number. `1` indicates that the dimension remains unchanged. The number of elements in *repeats* must be greater than or equals to the number of dimensions in *self* . When the number of dimensions of *self* is less than the number of elements of *repeats* , *self* is broadcasted to the number of dimensions with the same number of elements of *repeats* (as shown in the example).

Returns

Tensor, the new Tensor after the element is copied from the specified number of repetitions.

Raises

- **RuntimeError** –If the number of elements of *repeats* is less than the number of dimensions of *self* . Or *repeats* has negative element.
- **RuntimeError** –If the number of elements of *repeats* or the number of dimensions of *self* is larger than 8.
- **TypeError** –If type of *repeats* is unsupported.

See also:

- *mindspore.Tensor.reshape()*: Give a new shape to a tensor without changing its data.
- *mindspore.Tensor.resize()*: Changes shape and size of tensor in-place.
- *mindspore.Tensor.repeat_interleave()*: Repeats each element on the specified axis of a Tensor based on the specified number of times.
- *mindspore.Tensor.tile()*: Repeats a Tensor on each dimension for a specified number of times. And there is no requirement on the number of parameters *repeats* .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor
>>> a = Tensor([[0, 1, 2], [3, 4, 5]])
>>> print(a.repeat([3, 2]))
[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]]
>>> print(a.repeat(repeats=(2, 1, 3)))  # a is treated as a shape [1, 2, 3]
[[[0 1 2 0 1 2 0 1 2]
  [3 4 5 3 4 5 3 4 5]]
 [[0 1 2 0 1 2 0 1 2]
  [3 4 5 3 4 5 3 4 5]]]
```

mindspore.Tensor.repeat_interleave

`Tensor.`**`repeat_interleave`**(*repeats*, *dim=None*, *, *output_size=None*) → *Tensor*

Repeat elements of a tensor along a dim, like *mindspore.numpy.repeat()*.

> **Warning:** Only support on Atlas A2 training series.

Note: The self tensor to repeat values for. Must be of type: float16, float32, int8, uint8, int16, int32, or int64.

Parameters

- **repeats** (*Union[int, tuple, list, Tensor]*) –The number of times to repeat, must be positive.

- **dim** (*int, optional*) –The dim along which to repeat, Default: `None`. if dim is None, the self Tensor will be flattened and the output will alse be flattened.

Keyword Arguments

output_size (*int, optional*) –Total output size for the given axis (e.g. sum of repeats), Default: `None`.

Returns

One tensor with values repeated along the specified dim. If self has shape ($s1, s2, ..., sn$) and dim is i, the output will have shape ($s1, s2, ..., si * repeats, ..., sn$). The output type will be the same as the type of *self*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input1 = Tensor(np.array([[0, 1, 2], [3, 4, 5]]), mindspore.int32)
>>> output1 = input1.repeat_interleave(repeats=2, dim=0, output_size=None)
>>> input2 = Tensor(np.array([[1, 2], [3, 4]]), mindspore.int32)
>>> output2 = input2.repeat_interleave(Tensor(np.array([1, 2])), dim=0, output_size=None)
>>> print(output1)
>>> print(output2)
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
[[1 2]
 [3 4]
 [3 4]]
```

`Tensor.`**repeat_interleave**(*repeats*, *dim=None*) → *Tensor*

Repeat elements of a tensor along an dim, like *mindspore.numpy.repeat()*.

Note: The tensor to repeat values for. Must be of type: float16, float32, int8, uint8, int16, int32, or int64.

Parameters

- **repeats** (*Union[int, tuple, list, Tensor]*) –The number of times to repeat, must be positive.
- **dim** (*int, optional*) –The dim along which to repeat, Default: `None`. if dim is None, the self Tensor will be flattened and the output will alse be flattened.

Returns

One tensor with values repeated along the specified dim. If self has shape ($s1, s2, ..., sn$) and dim is i, the output will have shape ($s1, s2, ..., si * repeats, ..., sn$). The output type will be the same as the type of *self*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input1 = Tensor(np.array([[0, 1, 2], [3, 4, 5]]), mindspore.int32)
>>> output1 = input1.repeat_interleave(repeats=2, dim=0)
>>> input2 = Tensor(np.array([[1, 2], [3, 4]]), mindspore.int32)
>>> output2 = input2.repeat_interleave(Tensor(np.array([1, 2])), dim=0)
>>> print(output1)
>>> print(output2)
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
[[1 2]
 [3 4]
 [3 4]]
```

mindspore.Tensor.reshape

Tensor.**reshape**(**shape*) → *Tensor*

For details, please refer to *mindspore.ops.reshape()*.

mindspore.Tensor.reshape_as

Tensor.**reshape_as**(*other*)

Change the shape of the Tensor to the shape of *other* without changing the data.

Parameters

other (Tensor) –The result tensor has the same shape as *other*.

Returns

Tensor, has the same shape as *other*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], dtype=ms.float32)
>>> y = Tensor(np.arange(6).reshape(3,2))
>>> output = x.reshape_as(y)
>>> print(output)
[[-0.1  0.3]
 [ 3.6  0.4]
 [ 0.5 -3.2]]
```

mindspore.Tensor.resize

Tensor.**resize**(**new_shape*)

Changes shape and size of tensor in-place.

If the shape of the new tensor is larger than the shape of the original tensor, the new tensor will be filled with 0. And if the shape of the new tensor is smaller than the shape of the original tensor, the new tensor is filled with the elements of the original tensor in order.

Note: Instead of changing the size of the input tensor and returns nothing as in numpy, this method returns a new Tensor with the input size. Numpy argument *refcheck* is not supported.

Parameters

new_shape (*Union[ints, tuple of ints]*) –Shape of resized tensor.

Returns

Tensor.

See also:

- *mindspore.Tensor.reshape()*: Give a new shape to a tensor without changing its data.
- *mindspore.Tensor.repeat()*: Repeat elements of a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = x.resize(3, 3)
>>> print(y)
[[1. 2. 3.]
[4. 5. 6.]
[0. 0. 0.]]
>>> y = x.resize(2, 2)
>>> print(y)
[[1. 2.]
[3. 4.]]
```

mindspore.Tensor.reverse

Tensor.**reverse**(*axis*)

For details, please refer to *mindspore.ops.flip()*. The *axis* parameter in *Tensor.reverse* is equivalent to the *dims* parameter in *mindspore.ops.flip()*.

mindspore.Tensor.reverse_sequence

Tensor.**reverse_sequence**(*seq_lengths*, *seq_dim=0*, *batch_dim=0*)

For details, please refer to *mindspore.ops.reverse_sequence()*.

mindspore.Tensor.roll

Tensor.**roll**(*shifts*, *dims*) → *Tensor*

For details, please refer to *mindspore.ops.roll()*.

mindspore.Tensor.rot90

Tensor.**rot90**(*k*, *dims*)

For details, please refer to *mindspore.ops.rot90()*.

mindspore.Tensor.round

Tensor.**round**(*decimals=0*) → *Tensor*

For details, please refer to *mindspore.ops.round()*.

mindspore.Tensor.rsqrt

Tensor.**rsqrt**() → *Tensor*

For details, please refer to *mindspore.ops.rsqrt()*.

mindspore.Tensor.scatter

Tensor.**scatter**(*dim*, *index*, *src*) → *Tensor*

Update the value in *src* to *self* according to the specified index. For a 3-D tensor, the output will be:

```
output[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0

output[i][index[i][j][k]][k] = src[i][j][k]  # if dim == 1

output[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

Note: The backward is supported only for the case *src.shape* == *index.shape* when *src* is a tensor. The rank of the input tensor *self* must be at least 1.

Parameters

- **dim** (*int*) −Which axis to scatter. Accepted range is [-r, r) where r = rank(self).
- **index** (Tensor) −The index to do update operation whose data must be positive number with type of int32 or int64. Same rank as *self* . And accepted range is [-s, s) where s is the size along axis.
- **src** (Tensor, *float*) −The data doing the update operation with *self*. Can be a tensor with the same data type as *self* or a float number to scatter.

Returns

Tensor, has the same shape and type as *self* .

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If rank of any of *self* , *index* and *src* is less than 1.
- **ValueError** −If the rank of *src* is not equal to the rank of *self* .
- **TypeError** −If the data types of *self* and *src* have different dtypes.
- **RuntimeError** −If *index* has negative elements.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = input.scatter(dim=1, index=index, src=src)
>>> print(out)
[[1. 2. 8. 4. 8.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = input.scatter(dim=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
 [0. 0. 0. 0. 0.]
 [4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0.]
 [7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = input.scatter(dim=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
 [4. 0. 5. 0. 6.]
 [7. 0. 8. 0. 9.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Tensor.**scatter**(*axis*, *index*, *src*) → *Tensor*

Update the value in *src* to *self* according to the specified index. Refer to *mindspore.ops.tensor_scatter_elements()* for more details.

Note: The backward is supported only for the case *src.shape* == *index.shape*. The rank of the input tensor *self* must be at least 1.

Parameters

- **axis** (*int*) −Which axis to scatter. Accepted range is [-r, r) where r = rank(self).
- **index** (Tensor) −The index to do update operation whose data must be positive number with type of int32 or int64. Same rank as *self* . And accepted range is [-s, s) where s is the size along axis.
- **src** (Tensor, *float*) −The data doing the update operation with *self*. Can be a tensor with the same data type as *self* or a float number to scatter.

Returns

Tensor, has the same shape and type as *self* .

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If rank of any of *self* , *index* and *src* is less than 1.
- **ValueError** −If the rank of *src* is not equal to the rank of *self* .
- **TypeError** −If the data types of *self* and *src* have different dtypes.
- **RuntimeError** −If *index* has negative elements.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = input.scatter(axis=1, index=index, src=src)
>>> print(out)
[[1. 2. 8. 4. 8.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = input.scatter(axis=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
 [0. 0. 0. 0. 0.]
 [4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0.]
 [7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
```

(continues on next page)

(continued from previous page)

```
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = input.scatter(axis=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
 [4. 0. 5. 0. 6.]
 [7. 0. 8. 0. 9.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

mindspore.Tensor.scatter_

Tensor.**scatter_**(*dim*, *index*, *src*) → *Tensor*

Update the value in *src* to update *self* according to the specified *index*.

Index the dimension *self* selected by *dim* using *index* , traverse the other dimensions in sequence, update the value of *src* to *self* , and return *self* .

This operator is the inverse of the in-place version of *mindspore.Tensor.gather()* .

This operation provides another three overloads to support parameter *reduce* and scalar value.

Here's an example using a 3-dimension tensor.

```
self[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0

self[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

Warning:

- If multiple indexes point to the same position in *self* , the final value of this position in *self* is uncertain.
- On Ascend, behavior is unpredictable when the value of *index* is not in the range *[-self.shape[dim], self.shape[dim])* in forward.
- This is an experimental API that is subject to change or deletion.

Note: The inverse gradient from *self* to *src* can be calculated only when the shape of src is the same as that of *index*.

Parameters

- **dim** (*int*) −Which axis to scatter. Accepted range is *[-r, r)* where *r* is the rank of *self* .
- **index** (Tensor) −The index to access *self* on the target axis specified by *dim* whose dtype must be int32 or int64. If it is an empty Tensor, no operations is performed and directly returns *self* . Otherwise, its rank must be the same as *self* and the value range of each element must be *[-s, s)* where *s* is the size of *self* along axis *dim* .
- **src** (Tensor) −The data to doing the update operation with *self* . It should have the same dtype and rank as *self* .

Returns

Tensor, the modified *self* itself.

Raises

- **TypeError** –If type of *self* , *index* or *src* is unsupported.
- **RuntimeError** –If *dim* is out of the range *[-r, r)* .
- **RuntimeError** –If rank of *self* is larger than 8.
- **RuntimeError** –If dtype of tensor *self* , *index* or *src* is unsupported.
- **RuntimeError** –If dtype of *self* is not equal to the dtype of *src* .
- **RuntimeError** –If *self* , *index*, or *src* have different ranks and *index* is not an empty tensor.
- **RuntimeError** –If there is a dimension *d* that makes *index.size(d) > src.size(d)* .
- **RuntimeError** –If there is a dimension *d* that makes *index.size(d) > self.size(d)* .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, int64, float32
>>> this_tensor = Tensor([[1, 2], [3, 4]], dtype=float32)
>>> index = Tensor([[1, 0], [1, 0]], dtype=int64)
>>> src = Tensor([[4, 3], [2, 1]], dtype=float32)
>>> this_tensor.scatter_(1, index, src)
>>> print(this_tensor)
[[3., 4.],
 [1., 2.]]
```

`Tensor.`**`scatter_`**(*dim*, *index*, *src*, ***, *reduce*) → *Tensor*

Update the value in *src* to update *self* according to the specified *index*.

Using the operation specified by *reduce* to index the dimension *self* selected by *dim* using *index* , traverse the other dimensions in sequence, accumulate or multiply the value of *src* to *self* , and return *self* .

This operator is the inverse of the in-place version of *mindspore.Tensor.gather()* .

Expect that the replacement operation changes to accumulation or multiplication based on the parameter *reduce*, other operations are the same as the overloaded function that accept *src* without the parameter *reduce* .

Here's an example using a 3-dimension tensor.

```
self[index[i][j][k]][j][k] += src[i][j][k]  # if dim == 0, reduce == "add"

self[i][j][index[i][j][k]] *= src[i][j][k]  # if dim == 2, reduce == "multiply"
```

Warning:

- If multiple indexes point to the same position in *self* , the final value of this position in *self* is uncertain.
- On Ascend, behavior is unpredictable when the value of *index* is not in the range *[-self.shape[dim], self.shape[dim])* in forward.
- This is an experimental API that is subject to change or deletion.

Note: This overload function does not support reverse gradient calculation and will return zeros if calculate gradient.

Parameters

- **dim** (*int*) −Which axis to scatter. Accepted range is *[-r, r)* where *r* is the rank of *self* .
- **index** (Tensor) −The index to access *self* on the target axis specified by *dim* whose dtype must be int32 or int64. If it is an empty Tensor, no operations is performed and directly returns *self* . Otherwise, its rank must be the same as *self* and the value range of each element must be *[-s, s)* where *s* is the size of *self* along axis *dim* .
- **src** (Tensor) −The data to doing the accumulate or multiply operation with *self* . It should have the same dtype and rank as *self* .

Keyword Arguments

reduce (*str*) −Reduce operation, supports `"add"` and `"multiply"` . When *reduce* is `"add"` , *src* is accumulated to *input* base on *index* . When *reduce* is `"multiply"` , *src* is multiplied to *input* base on *index* .

Returns

Tensor, the modified *self* itself.

Raises

- **TypeError** −If type of *self* , *index* or *src* is unsupported.
- **ValueError** −If *reduce* is a str but not `"add"` or `"multiply"` .
- **RuntimeError** −If *dim* is out of the range *[-r, r)* .
- **RuntimeError** −If rank of *self* is larger than 8.
- **RuntimeError** −If dtype of tensor *self* , *index* or *src* is unsupported.
- **RuntimeError** −If dtype of *self* is not equal to the dtype of *src* .
- **RuntimeError** −If *self* , *index*, or *src* have different ranks and *index* is not an empty tensor.
- **RuntimeError** −If there is a dimension *d* that makes *index.size(d)* > *src.size(d)* .
- **RuntimeError** −If there is a dimension *d* that makes *index.size(d)* > *self.size(d)* .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, int64, float32
>>> this_tensor = Tensor([[1, 2], [3, 4]], dtype=float32)
>>> index = Tensor([[1, 0], [1, 0]], dtype=int64)
>>> src = Tensor([[4, 3], [2, 1]], dtype=float32)
>>> this_tensor.scatter_(1, index, src, reduce='add')
>>> print(this_tensor)
[[4., 6.],
 [4., 6.]]
```

Tensor.**scatter_**(*dim*, *index*, *value*) → *Tensor*

Update the value *value* to update *self* according to the specified *index*.

Index the dimension *self* selected by *dim* using *index* , traverse the other dimensions in sequence, update the value *value* to *self* , and return *self* .

This operator is the inverse of the in-place version of `mindspore.Tensor.gather()` .

It can be considered that after the value is broadcasted as a Tensor whose shape and dtype are consistent with *self* , other operations are the same as the overloaded function that accept *src* without the parameter *reduce* .

Here's an example using a 3-dimension tensor.

```
self[index[i][j][k]][j][k] = value  # if dim == 0

self[i][j][index[i][j][k]] = value  # if dim == 2
```

Warning:

- If multiple indexes point to the same position in *self* , the final value of this position in *self* is uncertain.
- On Ascend, behavior is unpredictable when the value of *index* is not in the range *[-self.shape[dim], self.shape[dim])* in forward.
- This is an experimental API that is subject to change or deletion.

Parameters

- **dim** (*int*) −Which axis to scatter. Accepted range is *[-r, r)* where *r* is the rank of *self* .
- **index** (Tensor) −The index to access *self* on the target axis specified by *dim* whose dtype must be int32 or int64. If it is an empty Tensor, no operations is performed and directly returns *self* . Otherwise, its rank must be the same as *self* and the value range of each element must be *[-s, s)* where *s* is the size of *self* along axis *dim* .
- **value** (*int, float, bool*) −The data to doing the update operation with *self* . It can be considered as being broadcasted into a Tensor whose shape and dtype are the same as *self* , and then be regarded as *src* for calculation.

Returns

Tensor, the modified *self* itself.

Raises

- **TypeError** −If type of *self* , *index* or *value* is unsupported.
- **RuntimeError** −If *dim* is out of the range *[-r, r)* .
- **RuntimeError** −If rank of *self* is larger than 8.
- **RuntimeError** −If dtype of tensor *self* or *index* is unsupported.
- **RuntimeError** −If *index* is not an empty tensor and its rank is different from the rank of *self* .
- **RuntimeError** −If there is a dimension *d* that makes *index.size(d) > self.size(d)* .

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor, int64, float32
>>> this_tensor = Tensor([[1, 2], [3, 4]], dtype=float32)
>>> index = Tensor([[0], [1]], dtype=int64)
>>> this_tensor.scatter_(0, index, 10)
>>> print(this_tensor)
[[10., 2.],
 [10., 4.]]
```

Tensor.**scatter_**(*dim*, *index*, *value*, *, *reduce*) → *Tensor*

Update the value *value* to update *self* according to the specified *index*.

Using the operation specified by *reduce* to index the dimension *self* selected by *dim* using *index* , traverse the other dimensions in sequence, accumulate or multiply the value *value* to *self* , and return *self* .

This operator is the inverse of the in-place version of *mindspore.Tensor.gather()* .

Expect that the replacement operation changes to accumulation or multiplication based on the parameter *reduce*, other operations are the same as the overloaded function that accept *value* without the parameter *reduce* .

Here's an example using a 3-dimension tensor.

```
self[i][index[i][j][k]][k] += value  # if dim == 1, reduce == "add"

self[i][j][index[i][j][k]] *= value  # if dim == 2, reduce == "multiply"
```

Warning:

- If multiple indexes point to the same position in *self* , the final value of this position in *self* is uncertain.
- On Ascend, behavior is unpredictable when the value of *index* is not in the range *[-self.shape[dim], self.shape[dim])* in forward.
- This is an experimental API that is subject to change or deletion.

Note: This overload function does not support reverse gradient calculation and will return zeros if calculate gradient.

Parameters

- **dim** (*int*) −Which axis to scatter. Accepted range is *[-r, r)* where *r* is the rank of *self* .
- **index** (Tensor) −The index to access *self* on the target axis specified by *dim* whose dtype must be int32 or int64. If it is an empty Tensor, no operations is performed and directly returns *self* . Otherwise, its rank must be the same as *self* and the value range of each element must be *[-s, s)* where *s* is the size of *self* along axis *dim* .
- **value** (*int, float, bool*) −The data to doing the accumulate or multiply operation with *self* . It can be considered as being broadcasted into a Tensor whose shape and dtype are the same as *self* , and then be regarded as *src* for calculation.

Keyword Arguments

reduce (*str*) −Reduce operation, supports "add" and "multiply" . When *reduce* is "add" , *value* is accumulated to *input* base on *index* . When *reduce* is "multiply" , *value* is multiplied to *input* base on *index* .

Returns

Tensor, the modified *self* itself.

Raises

- **TypeError** –If type of *self* , *index* or *value* is unsupported.
- **ValueError** –If *reduce* is a str but not `"add"` or `"multiply"` .
- **RuntimeError** –If *dim* is out of the range *[-r, r)* .
- **RuntimeError** –If rank of *self* is larger than 8.
- **RuntimeError** –If dtype of tensor *self* or *index* is unsupported.
- **RuntimeError** –If *index* is not an empty tensor and its rank is different from the rank of *self* .
- **RuntimeError** –If there is a dimension *d* that makes *index.size(d)* > *self.size(d)* .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, int64, float32
>>> this_tensor = Tensor([[1, 2], [3, 4]], dtype=float32)
>>> index = Tensor([[0], [1]], dtype=int64)
>>> this_tensor.scatter_(0, index, 3, reduce="multiply")
>>> print(this_tensor)
[[3., 2.],
 [9., 4.]]
```

mindspore.Tensor.scatter_add

`Tensor.`**`scatter_add`**(*dim*, *index*, *src*) → *Tensor*

Add all elements in *src* to the index specified by *index* to *self* along dimension specified by *dim*. It takes three inputs *self*, *src* and *index* of the same rank r >= 1.

For a 3-D tensor, the operation updates input as follows:

```
self[index[i][j][k]][j][k] += src[i][j][k]  # if dim == 0

self[i][index[i][j][k]][k] += src[i][j][k]  # if dim == 1

self[i][j][index[i][j][k]] += src[i][j][k]  # if dim == 2
```

Note: The rank of this tensor *self* must be at least 1.

Parameters

- **dim** (*int*) –Which dim to scatter. Accepted range is [-r, r) where r = rank(*self*).
- **index** (Tensor) –The index of *self* to do scatter operation whose data type must be int32 or int64. Same rank as *self*. Except for the dimension specified by *dim*, the size of each dimension of *index* must be less than or equal to the size of the corresponding dimension of *self*.

- **src** (Tensor) −The tensor doing the scatter operation with *self*, has the same type as *self* and the size of each dimension must be greater than or equal to that of *index*.

Returns

Tensor, has the same shape and type as *self*.

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If anyone of the rank among *self*, *index* and *src* is less than 1.
- **ValueError** −If the rank of *self*, *index* and *src* is not the same.
- **ValueError** −The size of any dimension of *index* except the dimension specified by *dim* is greater than the size of the corresponding dimension of *self*.
- **ValueError** −If the size of any dimension of *src* is less than that of *index*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = input.scatter_add(dim=1, index=index, src=src)
>>> print(out)
[[1. 2. 11. 4. 13.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = input.scatter_add(dim=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
 [0. 0. 0. 0. 0.]
 [4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0.]
 [7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = input.scatter_add(dim=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
 [4. 0. 5. 0. 6.]
 [7. 0. 8. 0. 9.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Tensor.**scatter_add**(*indices*, *updates*) → *Tensor*

Creates a new tensor by adding the values from the positions in *self* indicated by *indices*, with values from *updates*. When multiple values are given for the same index, the updated result will be the sum of all values. This operation is almost equivalent to using ScatterNdAdd, except that the updates are applied on output *Tensor* instead of input *Parameter*.

The last axis of *indices* is the depth of each index vectors. For each index vector, there must be a corresponding value in *updates*. The shape of *updates* should be equal to the shape of *self[indices]*. For more details, see Examples.

$$output\left[indices\right] = input\_x + update$$

Note: The dimension of this tensor *self* must be no less than indices.shape[-1].

If some values of the *indices* are out of bound:

- On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor.
- On CPU, if some values of the *indices* are out of bound, raising an index error.
- On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.

Parameters

- **indices** (Tensor) −The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) −The tensor to update the input tensor, has the same type as input, and updates. And the shape should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Returns

Tensor, has the same shape and type as *self*.

Raises

- **TypeError** −If dtype of *indices* is neither int32 nor int64.
- **ValueError** −If length of shape of *self* is less than the last dimension of shape of *indices*.
- **RuntimeError** −If a value of *indices* is not in *self* on CPU backend.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> output = input_x.scatter_add(indices, updates)
>>> print(output)
[[ 3.1  0.3  3.6]
 [ 0.4  0.5 -3.2]]
```

mindspore.Tensor.scatter_add_

Tensor.**scatter_add_**(*dim*, *index*, *src*)

Add all elements in *src* to the index specified by *index* to *self* along dimension specified by *dim*, *scatter_add* is an in-place operation. The ranks of *self*, *index* and *src* must be greater or equal to 1.

For a 3-D tensor, the operation updates *self* as follows:

```
self[index[i][j][k]][j][k] += src[i][j][k]  # if dim == 0

self[i][index[i][j][k]][k] += src[i][j][k]  # if dim == 1

self[i][j][index[i][j][k]] += src[i][j][k]  # if dim == 2
```

Parameters

- **dim** (*int*) −Which dim to scatter. Accepted range is [-r, r) where r = rank(*self*).
- **index** (Tensor) −The index of *self* to do scatter operation whose data type must be int32 or int64. Same rank as *self*. Except for the dimension specified by *dim*, size of each dimension of *index* must be less than or equal to the size of the corresponding dimension of *self*.
- **src** (Tensor) −The tensor doing the scatter operation with *self*, has the same type as *self* and the size of each dimension must be greater than or equal to that of *index*.

Returns

Tensor, has the same shape and type as *self*.

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If anyone of the rank among *self*, *index* and *src* is less than 1.
- **ValueError** −If the ranks of *self*, *index* and *src* are not the same.
- **ValueError** −The size of any dimension of *index* except the dimension specified by *dim* is greater than the size of the corresponding dimension of *self*.
- **ValueError** −If the size of any dimension of *src* is less than that of *index*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = input.scatter_add_(1, index, src)
>>> print(out)
[[1. 2. 11. 4. 13.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
```

(continues on next page)

(continued from previous page)

```
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = input.scatter_add_(0, index, src)
>>> print(out)
[[1. 2. 3. 0. 0.]
[0. 0. 0. 0. 0.]
[4. 5. 6. 0. 0.]
[0. 0. 0. 0. 0.]
[7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = input.scatter_add_(1, index, src)
>>> print(out)
[[1. 0. 2. 0. 3.]
[4. 0. 5. 0. 6.]
[7. 0. 8. 0. 9.]
[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]]
```

mindspore.Tensor.scatter_div

Tensor.**scatter_div**(*indices*, *updates*)
For details, please refer to *mindspore.ops.scatter_div()*.

mindspore.Tensor.scatter_max

Tensor.**scatter_max**(*indices*, *updates*)
For details, please refer to *mindspore.ops.scatter_max()*.

mindspore.Tensor.scatter_min

Tensor.**scatter_min**(*indices*, *updates*)
For details, please refer to *mindspore.ops.scatter_min()*.

mindspore.Tensor.scatter_mul

Tensor.**scatter_mul**(*indices*, *updates*)
For details, please refer to *mindspore.ops.scatter_mul()*.

mindspore.Tensor.scatter_sub

Tensor.**scatter_sub**(*indices*, *updates*)
For details, please refer to *mindspore.ops.tensor_scatter_sub()*.

mindspore.Tensor.searchsorted

Tensor.**searchsorted**(*v*, *side='left'*, *sorter=None*)

For details, please refer to *mindspore.ops.searchsorted()*.

mindspore.Tensor.select

Tensor.**select**(*dim*, *index*) → *Tensor*

Slices the *self* tensor along the selected dimension at the given index.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **dim** (*int*) −the dimension to slice.
- **index** (*int*) −the index to select with.

Returns

Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> input = Tensor([[2, 3, 4, 5], [3, 2, 4, 5]])
>>> y = Tensor.select(input, 0, 0)
>>> print(y)
[2 3 4 5]
```

Tensor.**select**(*condition*, *y*) → *Tensor*

The conditional tensor determines whether the corresponding element in the output must be selected from *self* (if True) or *y* (if False) based on the value of each element.

It can be defined as:

$$out_i = \begin{cases} self_i, & \text{if } condition_i \\ y_i, & \text{otherwise} \end{cases}$$

Parameters

- **condition** (*Tensor[bool]*) −The condition tensor, decides which element is chosen. The shape is $(x_1, x_2, ..., x_N, ..., x_R)$.
- **y** (*Union[Tensor, int, float]*) −The second Tensor to be selected. If *y* is a Tensor, its shape should be or be broadcast to $(x_1, x_2, ..., x_N, ..., x_R)$. If *y* is int or float, it will be casted to int32 or float32, and broadcast to the same shape as *self*. There must be at least one Tensor between *self* and *y*.

Returns

Tensor, has the same shape as *condition*.

Raises

- **TypeError** –If y is not a Tensor, int or float.
- **ValueError** –The shape of inputs cannot be broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> # Both input are Tensor
>>> cond = Tensor([True, False])
>>> x = Tensor([2,3], mindspore.float32)
>>> y = Tensor([1,2], mindspore.float32)
>>> output = Tensor.select(x, cond, y)
>>> print(output)
[2. 2.]
```

mindspore.Tensor.select_scatter

`Tensor.`**`select_scatter`**(*src*, *axis*, *index*)

For details, please refer to *mindspore.ops.select_scatter()*.

mindspore.Tensor.set_const_arg

`Tensor.`**`set_const_arg`**(*const_arg=True*)

Specify whether the tensor is a constant when it is used for the argument of a network.

Parameters

const_arg (*bool*) –Whether the tensor is a constant when it is used for the argument of a network. Default: `True` .

Returns

Tensor, has been specified whether to be a const network argument.

Raises

TypeError –If *const_arg* is not a bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1,2,3],[4,5,6]], dtype=np.float32))
>>> x.set_const_arg(True)
```

mindspore.Tensor.sgn

Tensor.**sgn**()

For details, please refer to *mindspore.ops.sgn()*.

mindspore.Tensor.shape

property Tensor.**shape**

For details, please refer to *mindspore.ops.shape()*.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> print(x.shape)
(2, 2)
```

mindspore.Tensor.short

Tensor.**short**()

Return a copy of the tensor, cast to int16 type, equivalent to self.astype(mstype.int16). If the value in tensor is float or half, the decimal will be discarded. For details, please refer to *mindspore.Tensor.astype()*.

Returns

Tensor, converted to the *int16* dtype.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.array([1,2,3,4,5]), ms.int32)
>>> output = x.short()
>>> output
Tensor(shape=[5], dtype=Int16, value= [1, 2, 3, 4, 5])
```

mindspore.Tensor.sigmoid

Tensor.**sigmoid**() → *Tensor*
For details, please refer to *mindspore.ops.sigmoid()*.

mindspore.Tensor.sign

Tensor.**sign**()
For details, please refer to *mindspore.ops.sign()*.

mindspore.Tensor.signbit

Tensor.**signbit**()
For details, please refer to *mindspore.ops.signbit()*.

mindspore.Tensor.sin

Tensor.**sin**() → *Tensor*
For details, please refer to *mindspore.ops.sin()*.

mindspore.Tensor.sinc

Tensor.**sinc**() → *Tensor*
For details, please refer to *mindspore.ops.sinc()*.

mindspore.Tensor.sinh

Tensor.**sinh**() → *Tensor*
For details, please refer to *mindspore.ops.sinh()*.

mindspore.Tensor.size

property Tensor.**size**
For details, please refer to *mindspore.ops.size()*.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.size
>>> print(output)
4
```

mindspore.Tensor.slice_scatter

Tensor.**slice_scatter**(*src*, *axis=0*, *start=None*, *end=None*, *step=1*)
For details, please refer to *mindspore.ops.slice_scatter()*.

mindspore.Tensor.slogdet

Tensor.**slogdet**()
For details, please refer to *mindspore.ops.slogdet()*.

mindspore.Tensor.softmax

Tensor.**softmax**(*axis*, *dtype=None*)
For details, please refer to *mindspore.ops.softmax()*.

mindspore.Tensor.sort

Tensor.**sort**(*dim=- 1*, *descending=False*)
Sorts the elements of the self tensor along the given dimension in the specified order.

> **Warning:** Currently, the data types of float16, uint8, int8, int16, int32, int64 are well supported. If use float32, it may cause loss of accuracy.

Parameters

- **dim** (*int, optional*) – The dimension to sort along. Default: -1, means the last dimension.
- **descending** (*bool, optional*) – Controls the sort order. If *descending* is True, the elements are sorted in descending order, or else sorted in ascending order. Default: False .

Returns

- y1, a tensor whose values are the sorted values, with the same shape and data type as self.
- y2, a tensor that consists of the indices of the elements in the original self tensor. Data type is int64.

Raises

- **TypeError** – If *dim* is not an int.
- **TypeError** – If *descending* is not a bool.
- **TypeError** – If *self* not in float16, float32, uint8, int8, int16, int32, int64, bfloat16
- **TypeError** – If *stable* is not a bool.
- **ValueError** – If *dim* is not in range of [-len(self.shape), len(self.shape)).

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> output = x.sort(dim=-1)
>>> # The output below is based on the Ascend platform.
>>> print(output)
(Tensor(shape=[3, 3], dtype=Float16, value=
[[ 1.0000e+00,  2.0000e+00,  8.0000e+00],
[ 3.0000e+00,  5.0000e+00,  9.0000e+00],
[ 4.0000e+00,  6.0000e+00,  7.0000e+00]]), Tensor(shape=[3, 3], dtype=Int64, value=
[[2, 1, 0],
[2, 0, 1],
[0, 1, 2]]))
```

`Tensor.`**`sort`**(*axis=- 1*, *descending=False*)

Sorts the elements of the input tensor along the given dimension in the specified order.

Parameters

- **`axis`** (*int, optional*) – The dimension to sort along. Default: `-1`, means the last dimension. The Ascend backend only supports sorting the last dimension.
- **`descending`** (*bool, optional*) – Controls the sort order. If *descending* is True, the elements are sorted in descending order, or else sorted in ascending order. Default: `False` .

> **Warning:** Currently, the data types of float16, uint8, int8, int16, int32, int64 are well supported. If use float32, it may cause loss of accuracy.

Returns

- y1, a tensor whose values are the sorted values, with the same shape and data type as self.
- y2, a tensor that consists of the indices of the elements in the original self tensor. Data type is int32.

Raises

- **`TypeError`** – If *axis* is not an int.
- **`TypeError`** – If *descending* is not a bool.
- **`TypeError`** – If dtype of *self* is neither float16, float32, uint8, int8, int16, int32, int64.
- **`ValueError`** – If *axis* is not in range of [-len(self.shape), len(self.shape)).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> output = x.sort(axis=-1)
>>> # The output below is based on the Ascend platform.
>>> print(output)
(Tensor(shape=[3, 3], dtype=Float16, value=
[[ 1.0000e+00,  2.0000e+00,  8.0000e+00],
[ 3.0000e+00,  5.0000e+00,  9.0000e+00],
[ 4.0000e+00,  6.0000e+00,  7.0000e+00]]), Tensor(shape=[3, 3], dtype=Int32, value=
[[2, 1, 0],
[2, 0, 1],
[0, 1, 2]]))
```

mindspore.Tensor.split

Tensor.**split**(*split_size*, *dim=0*)

Splits the Tensor into chunks along the given dim.

Parameters

- **split_size** (*Union[int, tuple(int), list(int)]*) – If *split_size* is an int type, *tensor* will be split into equally sized chunks, each chunk with size *split_size*. Last chunk will be smaller than *split_size* if *tensor.shape[dim]* is not divisible by *split_size*. If *split_size* is a list type, then *tensor* will be split into len(split_size) chunks with sizes *split_size* along the given *dim*.
- **dim** (*int, optional*) – The dim along which to split. Default: `0` .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** – If argument *dim* is not int.
- **ValueError** – If argument *dim* is out of range of $[-tensor.ndim, tensor.ndim)$.
- **TypeError** – If each element in *split_size* is not integer.
- **TypeError** – If argument *split_size* is not int, tuple(int) or list(int).
- **ValueError** – The sum of *split_size* is not equal to x.shape[dim].

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = np.arange(9).astype("float32")
>>> output = Tensor.split(Tensor(input_x), 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

`Tensor.`**`split`**(*split_size_or_sections*, *axis=0*)

Splits the Tensor into chunks along the given axis.

Parameters

- **split_size_or_sections** (*Union[int, tuple(int), list(int)]*) –If *split_size_or_sections* is an int type, *tensor* will be split into equally sized chunks, each chunk with size *split_size_or_sections*. Last chunk will be smaller than *split_size_or_sections* if *tensor.shape[axis]* is not divisible by *split_size_or_sections*. If *split_size_or_sections* is a list type, then *tensor* will be split into len(split_size_or_sections) chunks with sizes *split_size_or_sections* along the given *axis*.
- **axis** (*int, optional*) –The axis along which to split. Default: `0` .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –If argument *axis* is not int.
- **ValueError** –If argument *axis* is out of range of $[-tensor.ndim, tensor.ndim)$.
- **TypeError** –If each element in *split_size_or_sections* is not integer.
- **TypeError** –If argument *split_size_or_sections* is not int, tuple(int) or list(int).
- **ValueError** –The sum of *split_size_or_sections* is not equal to x.shape[axis].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = np.arange(9).astype("float32")
>>> output = Tensor.split(Tensor(input_x), 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

(continues on next page)

(continued from previous page)

mindspore.Tensor.sqrt

Tensor.**sqrt**() → *Tensor*
For details, please refer to *mindspore.ops.sqrt()*.

mindspore.Tensor.square

Tensor.**square**() → *Tensor*
For details, please refer to *mindspore.ops.square()*.

mindspore.Tensor.squeeze

Tensor.**squeeze**(*axis=None*)
For details, please refer to *mindspore.ops.squeeze()*.

mindspore.Tensor.std

Tensor.**std**(*axis=None*, *ddof=0*, *keepdims=False*) → *Tensor*
For details, please refer to *mindspore.ops.std()*.

Tensor.**std**(*dim=None*, ***, *correction=1*, *keepdim=False*) → *Tensor*

Calculates the standard deviation over the dimensions specified by *dim*. *dim* can be a single dimension, list of dimensions, or None to reduce over all dimensions.

The standard deviation (σ) is calculated as:

$$\sigma = \sqrt{\frac{1}{N - \delta N} \sum_{j-1}^{N-1} \left(self_{ij} - \overline{x_i}\right)^2}$$

where x is the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

dim (*None, int, tuple(int), optional*) – The dimension or dimensions to reduce. Defaults to None. If None, all dimensions are reduced.

Keyword Arguments

- **correction** (int, *optional*) – The difference between the sample size and sample degrees of freedom. Defaults to Bessel's correction. Defaults to 1.
- **keepdim** (bool, *optional*) – Whether the output tensor has dim retained or not. If True , keep these reduced dimensions and the length is 1. If False, don't keep these dimensions. Defaults to False.

Returns

Tensor, the standard deviation. Suppose the shape of *self* is $(x_0, x_1, ..., x_R)$:

- If *dim* is () and *keepdim* is set to `False` , returns a 0-D Tensor, indicating the standard deviation of all elements in *self*.
- If *dim* is int, e.g. `1` and *keepdim* is set to `False` , then the returned Tensor has shape $(x_0, x_2, ..., x_R)$.
- If *dim* is tuple(int) or list(int), e.g. `(1, 2)` and *keepdim* is set to `False` , then the returned Tensor has shape $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *self* is not a Tensor.
- **TypeError** –If *self* is not in bfloat16, float16, float32.
- **TypeError** –If *dim* is not one of the followings: None, int, tuple.
- **TypeError** –If *correction* is not an int.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range $[-self.ndim, self.ndim)$.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import mint, Tensor
>>> input = Tensor(np.array([[1, 2, 3], [-1, 1, 4]]).astype(np.float32))
>>> output = input.std(dim=1, correction=1, keepdim=False)
>>> print(output)
[1.      2.5166113]
```

mindspore.Tensor.storage_offset

`Tensor.`**`storage_offset`**`()`

Tensor's offset in the underlying storage in terms of the number of storage elements.

Returns

int, tensor's offset in the underlying storage in terms of number of storage elements.

Examples

```
>>> import mindspore as ms
>>> x = ms.Tensor([1, 2, 3, 4, 5], dtype=ms.float32)
>>> ret = x.storage_offset()
>>> print(ret)
0
```

mindspore.Tensor.stride

Tensor.**stride**(*dim=None*)

The stride to jump from one element to the next in the input dim. When no parameters are passed in, a list of stride for all dimensions is returned.

Parameters

dim (*int, optional*) – The dim of stride from one element to the next. Default: None.

Returns

Int, returns the step size necessary to jump from one element to the next in the specified dimension.

Raises

TypeError – *dim* is not an int.

Examples

```
>>> import mindspore as ms
>>> x = ms.Tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]], dtype=ms.float32)
>>> x.stride()
[5, 1]
```

mindspore.Tensor.strides

property Tensor.**strides**

Return the tuple of bytes to step in each dimension when traversing a tensor.

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]), dtype=mstype.int64)
>>> output = x.strides
>>> print(output)
(16, 8)
```

mindspore.Tensor.sub

Tensor.**sub**(*other*, *, *alpha=1*) → *Tensor*

Subtracts scaled other value from self Tensor.

$$out_i = self_i - alpha \times other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs and alpha comply with the implicit type conversion rules to make the data types consistent.

Parameters

other (*Union[Tensor, number.Number, bool]*) – The second self, is a number.Number or a bool or a tensor whose data type is number or bool_.

Keyword Arguments

alpha (*number.Number, optional*) – A scaling factor applied to *other*, default 1.

Returns

Tensor with a shape that is the same as the broadcasted shape of the self *self* and *other*, and the data type is the one with higher precision or higher digits among the two inputs and alpha.

Raises

- **TypeError** – If the type of *other* or *alpha* is not one of the following: Tensor, number.Number, bool.
- **TypeError** – If *alpha* is of type float but *self* and *other* are not of type float.
- **TypeError** – If *alpha* is of type bool but *self* and *other* are not of type bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> y = Tensor(1, mindspore.int32)
>>> output = Tensor.sub(x, y, alpha=0.5)
>>> print(output)
[3.5 4.5 5.5]
>>> # the data type of x is float32, the data type of y is int32,
>>> # alpha is a float, and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

Tensor.**sub**(*y*) → *Tensor*

For details, please refer to *mindspore.ops.sub()* .

mindspore.Tensor.sub_

Tensor.**sub_**(*other*, *, *alpha=1*) → *Tensor*

For details, please refer to *mindspore.mint.sub()*.

mindspore.Tensor.subtract

Tensor.**subtract**(*other*, *, *alpha=1*) → *Tensor*

This interface is deprecated from version 2.4 and will be removed in a future version.

mindspore.Tensor.sum

Tensor.**sum**(*dim=None*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Calculate sum of Tensor elements over a given dim.

Note: The *dim* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **dim** (*Union[None, int, tuple(int), list(int), Tensor], optional*) –Dimensions along which a sum is performed. If `None` , sum all the elements of the self tensor. If the *dim* is a tuple or list of ints, a sum is performed on all the dimensions specified in the tuple. Must be in the range $[-self.ndim, self.ndim)$. Default: `None` .
- **keepdim** (*bool, optional*) –Whether the output tensor has *dim* retained or not. If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: `None` .

Returns

A Tensor, sum of elements over a given *dim* in *self*.

Raises

- **TypeError** –If *dim* is not an int, tulpe(int), list(int), Tensor or None.
- **ValueError** –If *dim* is not in the range $[-self.ndim, self.ndim)$.
- **TypeError** –If *keepdim* is not a bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mstype.float32)
>>> out = Tensor.sum(x)
>>> print(out)
270.0
>>> out = Tensor.sum(x, dim=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(out)
[[ 6. 12. 18.]
[24. 30. 36.]
[42. 48. 54.]]
>>> out = Tensor.sum(x, dim=2, keepdim=True)
>>> print(out)
[[[ 6.]
[12.]
[18.]]
[[24.]
[30.]
[36.]]
[[42.]
[48.]
[54.]]]
```

`Tensor.`**sum**(*axis=None*, *dtype=None*, *keepdims=False*, *initial=None*) → *Tensor*

Return sum of tensor elements over a given axis.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **axis** (*Union[None, int, tuple(int), list(int), Tensor], optional*) – Axis or axes along which a sum is performed. Default: `None` . If `None` , sum all the elements of the self tensor. If the *axis* is negative, it counts from the last to the first *axis*. If the *axis* is a tuple or list of ints, a sum is performed on all the axes specified in the tuple or list instead of a single *axis* or all the axes as before.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.
- **keepdims** (*bool, optional*) – If this is set to `True` , the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the self array. If the default value is passed, then *keepdims* will not be passed through to the sum method of sub-classes of ndarray, however any non-default value will be. If the sub-class method does not implement *keepdims* any exceptions will be raised. Default: `False` .
- **initial** (*scalar, optional*) – Starting value for the sum. Default: `None` .

Returns

Tensor. A tensor with the same shape as self, with the specified *axis* removed. If the self tensor is a 0-d array, or if the *axis* is `None` , a scalar is returned.

Raises

- **TypeError** – If self is not array_like, or *axis* is not int, tuple of ints, list of ints or Tensor, or *keepdims* is not integer, or *initial* is not scalar.
- **ValueError** – If any *axis* is out of range or duplicate axes exist.

See also:

- *mindspore.Tensor.cumsum()*: Return the cumulative sum of the elements along a given *axis*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.array([-1, 0, 1]).astype(np.float32))
>>> print(input_x.sum())
0.0
>>> input_x = Tensor(np.arange(10).reshape(2, 5).astype(np.float32))
>>> print(input_x.sum(axis=1))
[10. 35.]
```

mindspore.Tensor.sum_to_size

Tensor.**sum_to_size**(**size*)

Sum self Tensor to the *size*. *size* must be expandable to the Tensor size.

Parameters

size (*Union[tuple(int), int]*) – The expected shape of output Tensor.

Returns

Tensor, the sum result of self Tensor according to the *size*.

Raises

ValueError – If *size* is not expandable to the size of self Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.random.randn(3, 3, 3, 3, 3, 3), mindspore.float32)
>>> output = x.sum_to_size((1, 3, 1, 3))
>>> print(output.shape)
(1, 3, 1, 3)
```

mindspore.Tensor.svd

Tensor.**svd**(*full_matrices=False*, *compute_uv=True*)

For details, please refer to *mindspore.ops.svd()*.

mindspore.Tensor.swapaxes

Tensor.**swapaxes**(*axis0*, *axis1*)
For details, please refer to *mindspore.ops.swapaxes()*.

mindspore.Tensor.swapdims

Tensor.**swapdims**(*dim0*, *dim1*)
For details, please refer to *mindspore.ops.swapdims()*.

mindspore.Tensor.T

property Tensor.**T**
Return the transposed tensor.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.array([[1, 2], [3, 4]]))
>>> output = x.T
>>> print(output)
[[1 3]
[2 4]]
```

mindspore.Tensor.take

Tensor.**take**(*indices*, *axis=None*, *mode='clip'*) → *Tensor*
Takes elements from a tensor along an axis.

Parameters

- **indices** (Tensor) –The indices with shape $(Nj...)$ of the values to extract.
- **axis** (*int, optional*) –The axis over which to select values. By default, the flattened input tensor is used. Default: None .
- **mode** (*str, optional*) –Support 'raise', 'wrap', 'clip'.
  - raise: Raises an error;
  - wrap: Wraps around;
  - clip: Clips to the range. 'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

  Default: 'clip' .

Returns

Tensor, the indexed result.

Raises

ValueError –If *axis* is out of range, or *mode* has values other than ('raise', 'wrap', 'clip').

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> a = Tensor(np.array([4, 3, 5, 7, 6, 8]))
>>> indices = Tensor(np.array([0, 1, 4]))
>>> output = a.take(indices)
>>> print(output)
[4 3 6]
```

`Tensor.`**`take`**(*index*) → *Tensor*

Select the self element at the given index.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

index (*LongTensor*) – The index tensor of self tensor.

Returns

Tensor, has the same data type as index tensor.

Raises

TypeError – If the dtype of *index* is not long type.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> input = Tensor([[4, 3, 5],[6, 7, 8]], ms.float32)
>>> index = Tensor([0, 2, 5], ms.int64)
>>> output = input.take(index)
>>> print(output)
[4, 5, 8]
```

mindspore.Tensor.tan

`Tensor.`**`tan`**() → *Tensor*

For details, please refer to *mindspore.ops.tan()*.

mindspore.Tensor.tanh

Tensor.**tanh**() → *Tensor*

For details, please refer to *mindspore.ops.tanh()*.

mindspore.Tensor.tensor_split

Tensor.**tensor_split**(*indices_or_sections*, *axis=0*)

For details, please refer to *mindspore.ops.tensor_split()*.

mindspore.Tensor.tile

Tensor.**tile**(*dims*) → *Tensor*

Replicates an tensor with given dims times.

Note: On Ascend, the number of *dims* should not exceed 8, and currently does not support scenarios where more than 4 dimensions are repeated simultaneously.

Parameters

dims (*tuple[int]*) –The parameter that specifies the number of replications, the parameter type is tuple, and the data type is int, i.e., $(y_1, y_2, ..., y_S)$. Only constant value is allowed.

Returns

Tensor, has the same data type as the *self*. Suppose the length of *dims* is *d*, the dimension of *self* is *self.dim*, and the shape of *self* is $(x_1, x_2, ..., x_S)$.

- If *self.dim* = *d*, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * y_1, x_2 * y_2, ..., x_S * y_S)$.
- If *self.dim* < *d*, prepend 1 to the shape of *self* until their lengths are consistent. Such as set the shape of *self* as $(1, ..., x_1, x_2, ..., x_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(1 * y_1, ..., x_R * y_R, x_S * y_S)$.
- If *self.dim* > *d*, prepend 1 to *dims* until their lengths are consistent. Such as set the *dims* as $(1, ..., y_1, y_2, ..., y_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * 1, ..., x_R * y_R, x_S * y_S)$.

Raises

- **TypeError** –If *dims* is not a tuple or not all elements are int.
- **ValueError** –If not all elements of *dims* are greater than or equal to 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[1, 2], [3, 4]]), mindspore.float32)
>>> dims = (2, 3)
>>> output = input.tile(dims)
>>> print(output)
[[1.  2.  1.  2.  1.  2.]
 [3.  4.  3.  4.  3.  4.]
 [1.  2.  1.  2.  1.  2.]
 [3.  4.  3.  4.  3.  4.]]
>>> dims = (2, 3, 2)
>>> output = input.tile(dims)
>>> print(output)
[[[1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]]
 [[1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]]]
```

`Tensor.tile`(*reps*) → *Tensor*

For more details, please refer to *mindspore.ops.tile()*. The parameter *reps* in the current interface and the parameter *dims* in the detail reference interface are actually the same parameter.

mindspore.Tensor.to

`Tensor.to`(*dtype*)

Performs tensor dtype conversion.

Note:

- If the *self* Tensor already has the correct *mindspore.dtype*, then self is returned. Otherwise, the returned tensor is a copy of *self* with the desired mindspore.dtype.
- When converting complex numbers to boolean type, the imaginary part of the complex number is not taken into account. As long as the real part is non-zero, it returns True; otherwise, it returns False.

Parameters

dtype (*dtype.Number*) – The valid data type of the output tensor. Only constant value is allowed.

Returns

Tensor, converted to the specified *dtype*.

Raises

TypeError – If *dtype* is not a Number.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_np = np.random.randn(2, 3, 4, 5).astype(np.float32)
>>> input_x = Tensor(input_np)
>>> dtype = mindspore.int32
>>> output = input_x.to(dtype)
>>> print(output.dtype)
Int32
```

mindspore.Tensor.to_coo

`Tensor.to_coo()`

Convert a Tensor to COOTensor.

Note: Only 2-D tensor is supported for now.

Returns

COOTensor, a sparse representation of the original dense tensor, containing the following parts.

- indices (Tensor): 2-D integer tensor, indicates the positions of *values* of the dense tensor.
- values (Tensor): 1-D tensor, indicates the non-zero values of the dense tensor.
- shape (tuple(int)): the shape of the COOTensor, is the same as the original dense tensor.

Raises

ValueError – If input tensor is not 2-D.

Supported Platforms:

`GPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1,  0], [-5, 0]]), mindspore.float32)
>>> output = x.to_coo()
>>> print(output.indices, output.values, output.shape)
[[0 0]
 [1 0]] [ 1. -5.] (2, 2)
```

mindspore.Tensor.to_csr

Tensor.**to_csr**()

Convert a Tensor to CSRTensor.

Note: Only 2-D tensor is supported for now.

Returns

CSRTensor, a sparse representation of the original dense tensor, containing the following parts.

- indptr (Tensor): 1-D integer tensor, indicates the start and end point for *values* in each row.
- indices (Tensor): 1-D integer tensor, indicates the column positions of all non-zero values of the input.
- values (Tensor): 1-D tensor, indicates the non-zero values of the dense tensor.
- shape (tuple(int)): the shape of the CSRTensor, is the same as the original dense tensor.

Raises

ValueError – If input tensor is not 2-D.

Supported Platforms:

GPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([[1,  0], [-5, 0]]), mindspore.float32)
>>> output = x.to_csr()
>>> print(output.indptr, output.indices, output.values, output.shape)
[0 1 2] [0 0] [ 1. -5.] (2, 2)
```

mindspore.Tensor.tolist

Tensor.**tolist**()

Convert a Tensor to List. If the input is Tensor scalar, a Python scalar will be returned.

Returns

List or Python scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> x = ms.Tensor([[1, 2, 3], [4, 5, 6]])
>>> out1 = x.tolist()
>>> print(out1)
[[1, 2, 3], [4, 5, 6]]
>>> out2 = x[0][0].tolist()
>>> print(out2)
1
```

mindspore.Tensor.topk

Tensor.**topk**(*k*, *dim=- 1*, *largest=True*, *sorted=True*)

Finds the *k* largest or smallest element along the given dimension and returns its value and corresponding index.

Warning:

- Due to different memory layout and traversal methods on different platforms, the display order of calculation results may be inconsistent when *sorted* is False.

If the *self* is a one-dimensional Tensor, finds the *k* largest or smallest entries in the Tensor, and outputs its value and index as a Tensor. *values[k]* is the *k* largest item in *self*, and its index is *indices[k]* .

For a multi-dimensional matrix, calculates the first or last *k* entries in a given dimension, therefore:

$$values.shape = indices.shape$$

If the two compared elements are the same, the one with the smaller index value is returned first.

Parameters

- **k** (*int*) –The number of top or bottom elements to be computed along the last dimension.
- **dim** (*int, optional*) –The dimension to sort along. Default: `-1` .
- **largest** (*bool, optional*) –If largest is `False` then the k smallest elements are returned. Default: `True` .
- **sorted** (*bool, optional*) –If `True` , the obtained elements will be sorted by the values in descending order or ascending order according to *largest*. If `False` , the obtained elements will not be sorted. Default: `True` .

Returns

A tuple consisting of *values* and *indices*.

- values (Tensor) - The *k* largest or smallest elements in each slice of the given dimension.
- indices (Tensor) - The indices of values within the last dimension of self.

Raises

- **TypeError** – If *sorted* is not a bool.
- **TypeError** – If *k* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> x = ms.Tensor([[0.5368, 0.2447, 0.4302, 0.9673],
...                [0.4388, 0.6525, 0.4685, 0.1868],
...                [0.3563, 0.5152, 0.9675, 0.8230]], dtype=ms.float32)
>>> output = Tensor.topk(x, 2, dim=1)
>>> print(output)
(Tensor(shape=[3, 2], dtype=Float32, value=
[[ 9.67299998e-01,  5.36800027e-01],
 [ 6.52499974e-01,  4.68499988e-01],
 [ 9.67499971e-01,  8.23000014e-01]]), Tensor(shape=[3, 2], dtype=Int32, value=
[[3, 0],
 [1, 2],
 [2, 3]]))
>>> output2 = Tensor.topk(x, 2, dim=1, largest=False)
>>> print(output2)
(Tensor(shape=[3, 2], dtype=Float32, value=
[[ 2.44700000e-01,  4.30200011e-01],
 [ 1.86800003e-01,  4.38800007e-01],
 [ 3.56299996e-01,  5.15200019e-01]]), Tensor(shape=[3, 2], dtype=Int32, value=
[[1, 2],
 [3, 0],
 [0, 1]]))
```

`Tensor.`**`topk`**(*k*, *dim=None*, *largest=True*, *sorted=True*)

For more details, please refer to *mindspore.ops.topk()*.

mindspore.Tensor.trace

`Tensor.`**`trace`**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*)

Return the sum along diagonals of the tensor.

Parameters

- **offset** (*int, optional*) – Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal.

- **axis1** (*int, optional*) −Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
- **axis2** (*int, optional*) −Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis.
- **dtype** (*mindspore.dtype*, optional) −defaults to None. Overrides the dtype of the output Tensor.

Returns

Tensor, the sum along diagonals.

Raises

ValueError −If the input tensor has less than two dimensions.

See also:

- *mindspore.Tensor.diagonal()*: Return specified diagonals.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.eye(3, dtype=np.float32))
>>> print(x.trace())
3.0
```

mindspore.Tensor.transpose

`Tensor.`**transpose**(*dim0*, *dim1*) → *Tensor*

Interchange two axes of a tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **dim0** (*int*) −Specifies the first dimension to be transposed.
- **dim1** (*int*) −Specifies the second dimension to be transposed.

Returns

Transposed tensor, has the same data type as *self*.

Raises

- **TypeError** −If *dim0* or *dim1* is not integer.
- **ValueError** −If *dim0* or *dim1* is not in the range of $[-ndim, ndim - 1]$.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.ones((2,3,4), dtype=np.float32))
>>> output = Tensor.transpose(input, 0, 2)
>>> print(output.shape)
(4, 3, 2)
```

`Tensor.`**`transpose`**`(*axes)` → *Tensor*

Permutes the dimensions of the self tensor according to self permutation.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector please refer to *mindspore.ops.expand_dims()*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and a.shape is $(i[0], i[1], ...i[n-2], i[n-1])$, then a.transpose().shape is $(i[n-1], i[n-2], ...i[1], i[0])$.

Note: On GPU and CPU, if the value of *axes* is negative, its actual value is *axes[i] + rank(self)*.

Parameters

axes (*tuple[int]*) –The permutation to be converted. The elements in *axes* are composed of the indexes of each dimension of *self*. The length of *axes* and the shape of *self* must be the same. Only constant value is allowed. Must be in the range [-rank(self), rank(self)).

Returns

Tensor, the type of output tensor is the same as *self* and the shape of output tensor is decided by the shape of *self* and the value of *axes*.

Raises

- **TypeError** –If *axes* is not a tuple.
- **ValueError** –If length of shape of *self* is not equal to length of shape of *axes*.
- **ValueError** –If the same element exists in *axes*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input = Tensor(np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]), mindspore.
↪float32)
>>> axes = (0, 2, 1)
>>> output = Tensor.transpose(input, axes)
>>> print(output)
[[[ 1.  4.]
  [ 2.  5.]
  [ 3.  6.]]
 [[ 7. 10.]
```

(continues on next page)

(continued from previous page)

```
  [ 8. 11.]
  [ 9. 12.]]]
```

mindspore.Tensor.triangular_solve

Tensor.**triangular_solve**(*A*, *upper=True*, *transpose=False*, *unitriangular=False*)
For details, please refer to *mindspore.mint.triangular_solve()*.

mindspore.Tensor.tril

Tensor.**tril**(*diagonal=0*) → *Tensor*
For details, please refer to *mindspore.ops.tril()*.

mindspore.Tensor.triu

Tensor.**triu**(*diagonal=0*) → *Tensor*
For details, please refer to *mindspore.ops.triu()*.

mindspore.Tensor.true_divide

Tensor.**true_divide**()
Alias for Tensor.div() with $rounding\_mode = None$. For details, please refer to *mindspore.ops.div()*.

mindspore.Tensor.trunc

Tensor.**trunc**() → *Tensor*
For details, please refer to *mindspore.ops.trunc()*.

mindspore.Tensor.type

Tensor.**type**(*dtype=None*)
Change the dtype of the Tensor to the *dtype* . Return the type if *dtype* is None .

Parameters

dtype (mindspore.dtype, *optional*) – The specified dtype of output tensor. Default: None.

Returns

Tensor or str. If *dtype* is None , return a str, which describes the dtype of Tensor. If *dtype* is not None , then return a Tensor, and the dtype of returned Tensor is *dtype* .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor([[1.2, 2], [3.4, 4]], dtype=mindspore.float32)
>>> print(x.type())
Float32
>>> print(x.type(dtype=mindspore.int32))
[[1 2]
 [3 4]]
```

mindspore.Tensor.type_as

Tensor.**type_as**(*other*)

Returns self tensor cast to the type of the with the input other tensor.

Warning: This is an experimental API that is subject to change or deletion.

Note: When converting complex numbers to boolean type, the imaginary part of the complex number is not taken into account. As long as the real part is non-zero, it returns True; otherwise, it returns False.

Parameters

other (Tensor) – The tensor whose data type is specified. The shape of tensor is $(x_0, x_1, ..., x_R)$.

Returns

Tensor, the shape of tensor is the same as *self*, $(x_0, x_1, ..., x_R)$.

Raises

TypeError – If *other* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_np = np.random.randn(2, 3, 4, 5).astype(np.float32)
>>> self = Tensor(input_np)
>>> other_np = np.random.randn(2, 3, 4).astype(np.int32)
>>> other = Tensor(other_np)
>>> output = self.type_as(other)
>>> print(output.dtype)
Int32
>>> print(output.shape)
(2, 3, 4, 5)
```

mindspore.Tensor.unbind

Tensor.**unbind**(*dim=0*) → *Tensor*

For details, please refer to *mindspore.ops.unbind()*.

mindspore.Tensor.unfold

Tensor.**unfold**(*kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

For details, please refer to *mindspore.ops.unfold()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

mindspore.Tensor.uniform

Tensor.**uniform**(*from_=0.*, *to=1.*, *generator=None*)

Generates random numbers that follows a uniform distribution within the half-open interval $[from\_, to)$.

$$P(x) = \frac{1}{to - from\_}$$

Parameters

- **from_** (*number*) –The lower bound of the interval.
- **to** (*number*) –The upper bound of the interval.
- **generator** (*Generator, optional*) –The random seed. Default: None.

Returns

Tensor, with the same shape as tensor.

Raises

TypeError –If *from_* is larger than *to*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.ops.ones((4, 2))
>>> generator = mindspore.Generator()
>>> generator.manual_seed(100)
>>> output = x.uniform(1., 2., generator)
>>> print(output.shape)
(4, 2)
```

mindspore.Tensor.uniform_

Tensor.**uniform_**(*from_=0*, *to=1*, ***, *generator=None*)

Update the *self* Tensor in place by generating random numbers sampled from uniform distribution in the half-open interval $[from\_, to)$.

$$P(x) = \frac{1}{to - from\_}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **from_** (*Union[number.Number, Tensor], optional*) –The lower bound of the uniform distribution, it can be a scalar value or a tensor of any dimension with a single element. Default: 0.
- **to** (*Union[number.Number, Tensor], optional*) –The upper bound of the uniform distribution, it can be a scalar value or a tensor of any dimension with a single element. Default: 1.

Keyword Arguments

generator (*mindspore.Generator*, optional) –a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.

Returns

Return *self* Tensor.

Raises

- **TypeError** –If *from_* or *to* is neither a number nor a Tensor.
- **TypeError** –If dtype of *from* or *to* is not one of: bool, int8, int16, int32, int64, uint8, float32, float64.
- **ValueError** –If *from_* or *to* is Tensor but contains multiple elements.
- **RuntimeError** –If *from_* is larger than *to*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.ops.ones((4, 2))
>>> generator = mindspore.Generator()
>>> generator.manual_seed(100)
>>> output = x.uniform_(1., 2., generator=generator)
>>> print(output.shape)
(4, 2)
```

mindspore.Tensor.unique

Tensor.**unique**(*sorted=True*, *return_inverse=False*, *return_counts=False*, *dim=None*)

Returns the unique elements of *self*.

when *return_inverse=True*, also return a tensor containing the index of each value of *self* corresponding to the output unique tensor. when *return_counts=True*, also return a tensor containing the number of occurrences for each unique value or tensor.

Parameters

- **sorted** (*bool, optional*) –Whether to sort the unique elements in ascending order before returning as output. Default: `True` .
- **return_inverse** (*bool, optional*) –Whether to also return the indices for where elements in *self* ended up in the returned unique list. Default: `False` .
- **return_counts** (*bool, optional*) –Whether to also return the counts for each unique element. Default: `False` .
- **dim** (*int, optional*) –the dimension to operate upon. If `None`, the unique of the flattened *self* is returned. Otherwise, each of the tensors indexed by the given dimension is treated as one of the elements to apply the unique operation upon. Default: `None` .

Returns

A tensor or a tuple of tensors containing some of tensor objects (*output*, *inverse_indices*, *counts*).

- **output** (Tensor) - The output tensor including the unique elements of *self*, it has same dtype as *self*.
- **inverse_indices** (Tensor, optional) - Return when `return_inverse` is True. It represents the indices for where elements in *self* map to in the output. When `dim` is `None`, it has same shape as *self*, otherwise, the shape is self.shape[dim].
- **counts** (Tensor, optional) - Return when `return_counts` is True. It represents the number of occurrences for each unique value or tensor. When `dim` is `None`, it has same shape as output, otherwise, the shape is output.shape(dim).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> x = Tensor(np.array([1, 2, 5, 2]), mindspore.int32)
>>> output = x.unique(return_inverse=True)
>>> print(output)
(Tensor(shape=[3], dtype=Int32, value= [1, 2, 5]), Tensor(shape=[4], dtype=Int64, value= [0,␣
↪1, 2, 1]))
>>> y = output[0]
>>> print(y)
[1 2 5]
>>> idx = output[1]
>>> print(idx)
[0 1 2 1]
```

mindspore.Tensor.unique_consecutive

Tensor.**unique_consecutive**(*return_inverse=False*, *return_counts=False*, *dim=None*)
For details, please refer to *mindspore.ops.unique_consecutive()*.

mindspore.Tensor.unsorted_segment_max

Tensor.**unsorted_segment_max**(*segment_ids*, *num_segments*)
For details, please refer to *mindspore.ops.unsorted_segment_max()*.

mindspore.Tensor.unsorted_segment_min

Tensor.**unsorted_segment_min**(*segment_ids*, *num_segments*)
For details, please refer to *mindspore.ops.unsorted_segment_min()*.

mindspore.Tensor.unsorted_segment_prod

Tensor.**unsorted_segment_prod**(*segment_ids*, *num_segments*)
For details, please refer to *mindspore.ops.unsorted_segment_prod()*.

mindspore.Tensor.unsqueeze

Tensor.**unsqueeze**(*dim*) → *Tensor*
For details, please refer to *mindspore.ops.unsqueeze()*.

mindspore.Tensor.var

Tensor.**var**(*axis=None*, *ddof=0*, *keepdims=False*) → *Tensor*
Compute the variance along the specified axis.

The variance is the average of the squared deviations from the mean, i.e., $var = mean(abs(x - x.mean()) **2)$.

Return the variance, which is computed for the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *dtype*, *out* and *where* are not supported.

Parameters

- **axis** (*Union[None, int, tuple(int)], optional*) −Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array. Default: `None` .
- **ddof** (*int, optional*) −Means Delta Degrees of Freedom. Default: `0` . The divisor used in calculations is $N - ddof$, where N represents the number of elements.
- **keepdims** (*bool, optional*) −Whether the output Tensor has dim retained or not. If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Returns

Variance tensor.

Raises

- **TypeError** –If *axis* is not one of the followings: None, int, tuple.
- **TypeError** –If *ddof* is not an int.
- **TypeError** –If *keepdims* is not a bool.
- **ValueError** –If *axis* is out of range $[-self.ndim, self.ndim)$.

See also:

- *mindspore.Tensor.mean()*: Reduce a dimension of a tensor by averaging all elements in the dimension.
- *mindspore.Tensor.std()*: Compute the standard deviation along the specified axis.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> input_x = Tensor(np.array([1., 2., 3., 4.], np.float32))
>>> output = input_x.var()
>>> print(output)
1.25
```

`Tensor.`**var**(*dim=None*, *, *correction=1*, *keepdim=False*) → *Tensor*

Calculates the variance over the dimensions specified by *dim*. *dim* can be a single dimension, list of dimensions, or None to reduce over all dimensions.

The variance (δ^2) is calculated as:

$$\delta^2 = \frac{1}{\max(0, N - \delta N)} \sum_{i=0}^{N-1} (x_i - \bar{x})^2$$

where x is the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction*.

Parameters

dim (*None*, *int*, *tuple(int)*, *optional*) –The dimension or dimensions to reduce. Defaults to `None`. If `None`, all dimensions are reduced.

Keyword Arguments

- **correction** (`int`, *optional*) –The difference between the sample size and sample degrees of freedom. Defaults to Bessel's correction. Defaults to `1`.
- **keepdim** (`bool`, *optional*) –Whether the output tensor has dim retained or not. If `True` , keep these reduced dimensions and the length is 1. If `False`, don't keep these dimensions. Defaults to `False`.

Returns

Tensor, the variance. Suppose the shape of *self* is $(x_0, x_1, ..., x_R)$:

- If *dim* is () and *keepdim* is set to `False` , returns a 0-D Tensor, indicating the variance of all elements in *self*.

- If *dim* is int, e.g. `1` and *keepdim* is set to `False` , then the returned Tensor has shape $(x_0, x_2, ..., x_R)$.
- If *dim* is tuple(int) or list(int), e.g. `(1, 2)` and *keepdim* is set to `False` , then the returned Tensor has shape $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *dim* is not one of the followings: None, int, list, tuple.
- **TypeError** –If *correction* is not an int.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range $[-self.ndim, self.ndim)$.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = Tensor([[8, 2, 1], [5, 9, 3], [4, 6, 7]], mindspore.float32)
>>> output = input_x.var(dim=0, correction=1, keepdim=True)
>>> print(output)
[[ 4.333333, 12.333333, 9.333333]]
```

mindspore.Tensor.view

`Tensor.`**`view`**`(*shape)`

Reshape the tensor according to the input shape. It's the same as *mindspore.Tensor.reshape()*, implemented by the underlying reshape operator.

Parameters

shape (*Union[tuple(int), int]*) –Dimension of the output tensor.

Returns

Tensor, which dimension is the input shape's value.

Examples

```
>>> from mindspore import Tensor
>>> import numpy as np
>>> a = Tensor(np.array([[1, 2, 3], [2, 3, 4]], dtype=np.float32))
>>> output = a.view((3, 2))
>>> print(output)
[[1. 2.]
[3. 2.]
[3. 4.]]
```

mindspore.Tensor.view_as

Tensor.**view_as**(*other*) → *Tensor*

View *self* Tensor as the same shape as *other* .

Parameters

other (Tensor) –The returned Tensor has the same shape as *other*.

Returns

Tensor, has the same shape as *other*.

Raises

TypeError –If *other* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> a = Tensor([[1, 2, 3], [2, 3, 4]], mstype.float32)
>>> b = Tensor([1, 1, 1, 1, 1, 1], mstype.float32)
>>> output = a.view_as(b)
>>> print(output)
[1. 2. 3. 2. 3. 4.]
```

mindspore.Tensor.vsplit

Tensor.**vsplit**(*indices_or_sections*)

For details, please refer to *mindspore.ops.vsplit()*.

mindspore.Tensor.where

Tensor.**where**(*condition*, *y*) → *Tensor*

For details, please refer to *mindspore.ops.where()*.

mindspore.Tensor.xdivy

Tensor.**xdivy**(*y*)

For details, please refer to *mindspore.ops.xdivy()*.

mindspore.Tensor.xlogy

Tensor.**xlogy**(*other*) → *Tensor*

For details, please refer to *mindspore.ops.xlogy()*.

mindspore.Tensor.zero_

Tensor.**zero_**()

Return a tensor filled with zeros.

> **Warning:** This is an experimental API that is subject to change or deletion.

Returns

Return a tensor. Fill self tensor with zeros.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> x = Tensor(np.array([2, 2]))
>>> output = x.zero_()
>>> print(output)
[0 0]
```

mindspore.tensor

mindspore.**tensor**(*input_data=None*, *dtype=None*, *shape=None*, *init=None*, *const_arg=False*)

Create a new Tensor in Cell.construct() or function decorated by @jit.

In graph mode, MindSpore would create a new Tensor object at runtime dynamically, based on the *dtype* argument.

Please refer to Creating and Using Tensor .

The difference between it and the Tensor class is that it adds Annotation which can prevent the generation of AnyType compared to the Tensor class.

The arguments and return values are the same as the Tensor class. Also see: *mindspore.Tensor*. internally to indicate the type of the Tensor currently being created,

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import jit, tensor
>>> @jit
... def func(x):
...     return tensor(x.asnumpy(), dtype=ms.float32)
>>> x = tensor([1, 2, 3])
>>> y = func(x)
>>> print(y)
[1. 2. 3.]
```

mindspore.COOTensor

class mindspore.**COOTensor**(*indices=None*, *values=None*, *shape=None*, *coo_tensor=None*)

A sparse representation of a set of nonzero elements from a tensor at given indices, where the indices indicate the position of each non-zero element.

For a tensor dense, its COOTensor(indices, values, shape) has *dense[indices[i]] = values[i]*.

For example, if indices is [[0, 1], [1, 2]], values is [1, 2], shape is (3, 4), then the dense representation of the sparse tensor will be:

```
[[0, 1, 0, 0],
 [0, 0, 2, 0],
 [0, 0, 0, 0]]
```

Common arithmetic operations include: addition (+), subtraction (-), multiplication (*), and division (/). For details about operations supported by *COOTensor*, see operators.

Warning:

- This is an experimental API that is subject to change or deletion.
- If use PyNative mode, set "export MS_PYNATIVE_CONFIG_STATIC_SHAPE=1".
- Currently, duplicate coordinates in the indices will not be coalesced. If the indices contain out-of-bound values, the result will be undefined.

Parameters

- **indices** (Tensor) −A 2-D integer Tensor of shape $(N, ndims)$, where N and $ndims$ are the number of *values* and number of dimensions in the COOTensor, respectively. Currently, $ndims$ must be 2. Default: None . Please make sure that the indices are in range of the given shape.
- **values** (Tensor) −A 1-D tensor of any type and shape (N), which supplies the values for each element in *indices*. Default: None .
- **shape** (*tuple(int)*) −An integer tuple of shape $(ndims)$, which specifies the dense_shape of the sparse tensor. Default: None .
- **coo_tensor** (COOTensor) −A COOTensor object. Default: None .

Returns

COOTensor, composed of *indices*, *values*, and *shape*.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> print(x.values)
[1. 2.]
>>> print(x.indices)
[[0 1]
 [1 2]]
>>> print(x.shape)
(3, 4)
```

abs()

Return absolute value element-wisely.

Returns

COOTensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1, 2], [1, 0, 2]], dtype=ms.int32)
>>> values = Tensor([1, -5, -4], dtype=ms.float32)
>>> shape = (3, 3)
>>> coo_tensor = COOTensor(indices.transpose(), values, shape)
>>> res = coo_tensor.abs()
>>> print(res.values)
[1. 5. 4.]
```

add(*other:* COOTensor, *thresh:* Tensor)

Return the sum with another COOTensor.

Parameters

- **other** (`COOTensor`) – the second SparseTensor to sum.
- **thresh** (`Tensor`) – A 0-D Tensor, represents the magnitude threshold that determines if an output value/index pair take space, Its dtype should match that of the values if they are real. If output's value is less than the *thresh*, it will vanish.

Returns

COOTensor, representing the sum.

Raises

- **ValueError** – If any input(self/other)'s indices's dim is not equal to 2.
- **ValueError** – If any input(self/other)'s values's dim is not equal to 1.

- **ValueError** –If any input(self/other)'s shape's dim is not equal to 1.
- **ValueError** –If thresh's dim is not equal to 0.
- **TypeError** –If any input(self/other)'s indices's type is not equal to int64.
- **TypeError** –If any input(self/other)'s shape's type is not equal to int64.
- **ValueError** –If any input(self/other)'s indices's length is not equal to its values's length.
- **TypeError** –If any input(self/other)'s values's type is not equal to anf of (int8/int16/int32/int64/float32/float64/complex64/complex128)
- **TypeError** –If thresh's type is not equal to anf of (int8/int16/int32/int64/float32/float64)
- **TypeError** –If self's indices's type is not equal to other's indices's type
- **TypeError** –If self's values's type is not equal to other's values's type
- **TypeError** –If self's shape's type is not equal to other's shape's type
- **TypeError** –If (self/other)'s value's type is not matched with thresh's type

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, COOTensor
>>> from mindspore import dtype as mstype
>>> indics0 = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values0 = Tensor([1, 2], dtype=mstype.int32)
>>> shape0 = (3, 4)
>>> input0 = COOTensor(indics0, values0, shape0)
>>> indics1 = Tensor([[0, 0], [1, 1]], dtype=mstype.int64)
>>> values1 = Tensor([3, 4], dtype=mstype.int32)
>>> shape1 = (3, 4)
>>> input1 = COOTensor(indics1, values1, shape1)
>>> thres = Tensor(0, dtype=mstype.int32)
>>> out = input0.add(input1, thres)
>>> print(out)
COOTensor(shape=[3, 4], dtype=Int32, indices=Tensor(shape=[4, 2], dtype=Int64, value=
[[0 0]
 [0 1]
 [1 1]
 [1 2]]), values=Tensor(shape=[4], dtype=Int32, value=[3 1 4 2]))
```

astype(*dtype: mstype*)

Return a copy of the COOTensor, cast its values to a specified type.

Parameters

dtype (Union[*mindspore.dtype*, numpy.dtype, str]) –Designated tensor dtype.

Returns

COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (3, 4)
>>> coo_tensor = COOTensor(indices, values, shape)
>>> print(coo_tensor.astype(ms.float64).dtype)
Float64
```

coalesce()

Returns a coalesced copy of an uncoalesced sparse tensor.

Returns

A COOTensor.

Supported Platforms:

GPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> x_indices = Tensor([[0, 0, 1], [1, 1, 2]], dtype=ms.int64)
>>> x_values = Tensor([1, 5, 4], dtype=ms.float32)
>>> x_shape = (3, 3)
>>> coo_tensor = COOTensor(x_indices.transpose(), x_values, x_shape)
>>> res = coo_tensor.coalesce()
>>> print(res)
COOTensor(shape=[3, 3], dtype=Float32, indices=Tensor(shape=[2, 2], dtype=Int64,
    value=[[0 1] [1 2]]), values=Tensor(shape=[2], dtype=Float32, value=[6.00000000e+00␣
↪4.00000000e+00]))
```

property dtype: mstype

Return the dtype of the values of COOTensor (*mindspore.dtype*).

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (3, 4)
>>> coo_tensor = COOTensor(indices, values, shape)
>>> print(coo_tensor.dtype)
Float32
```

property indices: mindspore.common.tensor.Tensor

Return COOTensor's indices.

property itemsize: int

Return the length of one tensor element in bytes.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float64)
>>> shape = (3, 4)
>>> coo_tensor = COOTensor(indices, values, shape)
>>> print(coo_tensor.itemsize)
8
```

property ndim: int
Return the number of tensor dimensions.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> coo_tensor = COOTensor(indices, values, (3, 4))
>>> print(coo_tensor.ndim)
2
```

property shape: Tuple[int, ...]
Return COOTensor's shape.

property size: int
Return the number of non-zero values.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1, 2], [1, 0, 2]], dtype=ms.int32)
>>> values = Tensor([1, 5, 4], dtype=ms.float32)
>>> shape = (3, 3)
>>> coo_tensor = COOTensor(indices.transpose(), values, shape)
>>> print(coo_tensor.size)
3
```

to_csr()
Converts COOTensor to CSRTensor.

Note: Currently only supports CPU backend with LLVM 12.0.1 installed.

Returns

CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.int32)
>>> shape = (3, 4)
>>> coo_tensor = COOTensor(indices, values, shape)
>>> print(coo_tensor.to_csr())
CSRTensor(shape=[3, 4], dtype=Int32, indptr=Tensor(shape=[4], dtype=Int32, value=[0 1 2␣
↪2]),
    indices=Tensor(shape=[2], dtype=Int32, value=[1 2]), values=Tensor(shape=[2],␣
↪dtype=Int32, value=[1 2]))
```

to_dense()

Converts COOTensor to Dense Tensor.

Returns

Tensor.

Supported Platforms:

GPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1, 2], [1, 0, 2]], dtype=ms.int32)
>>> values = Tensor([1, 5, 4], dtype=ms.float32)
>>> shape = (3, 3)
>>> coo_tensor = COOTensor(indices.transpose(), values, shape)
>>> print(coo_tensor.to_dense())
[[0. 1. 0.]
 [5. 0. 0.]
 [0. 0. 4.]]
```

to_tuple()

Return indices, values and shape as a tuple.

Returns

Tuple.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (3, 4)
>>> coo_tensor = COOTensor(indices, values, shape)
>>> print(coo_tensor.to_tuple())
(Tensor(shape=[2, 2], dtype=Int32, value=
[[0, 1],
 [1, 2]]), Tensor(shape=[2], dtype=Float32, value= [ 1.00000000e+00,  2.00000000e+00]),␣
↪(3, 4))
```

property values: mindspore.common.tensor.Tensor

Return COOTensor's non-zero values.

mindspore.CSRTensor

class mindspore.**CSRTensor**(*indptr=None*, *indices=None*, *values=None*, *shape=None*, *csr_tensor=None*)

Constructs a sparse tensor in CSR (Compressed Sparse Row) format, with specified values indicated by *values* and row and column positions indicated by *indptr* and *indices*.

For example, if indptr is [0, 2, 5, 6], indices is [0, 3, 1, 2, 4, 2], values is [1., 2., 3., 4., 5., 6.], shape is (3, 5), then the dense representation of the sparse tensor will be:

```
[[1., 0., 0., 2., 0.],
 [0., 3., 4., 0., 5.],
 [0., 0., 6., 0., 0.]]
```

The length of *indptr* should equal to *shape[0]+1*, where the elements should be equal or monotonically increasing and the maximum value should be equal to the number of non-zero values in the tensor. The length of *indices* and *values* should be equal to the number of non-zero values in the tensor. To be concrete, get the query indices of none-zero elements in every line according to *indptr*. Then get the column positions of none-zero elements in every line by looking up query indices in *indices*. Finally, get the actual values of none-zero elements in every line by looking up query indices in *values*.

In the former example, 'indptr' of [0, 2, 5, 6] represents that the indices of 0th row of the tensor origins from [0, 2), the indices of the 1st row of the tensor origins from [2, 5) and the 2nd row of the tensor origins from [5, 6). For example:

- The column positions of the non-zero elements of the 0th row in the tensor are provided by the [0, 2) elements in *indices* (i.e. [0, 3]) and the corresponding values are provided by the [0, 2) elements in *values* (i.e. [1., 2.]).
- The column positions of the non-zero elements of the 1st row in the tensor are provided by the [2, 5) elements in *indices* (i.e. [1, 2, 4]) and the corresponding values are provided by the [2, 5) elements in *values* (i.e. [3., 4., 5.]).
- The column positions of the non-zero elements of the 2nd row in the tensor are provided by the [5, 6) elements in *indices* (i.e. [2]) and the corresponding values are provided by the [5, 6) elements in *values* (i.e. [6.]).

Common arithmetic operations include: addition (+), subtraction (-), multiplication (*), and division (/). For details about operations supported by *CSRTensor*, see operators.

Warning:

- This is an experimental API that is subjected to change.
- If use PyNative mode, set "export MS_PYNATIVE_CONFIG_STATIC_SHAPE=1".

- If the values given by *indptr* or *indices* are invalid, the results may be undefined. Invalid values include when the length of *values* or *indices* exceeds the range indicated by *indptr*, and when the columns indicated by *indices* are repeated on the same row.

Parameters

- **indptr** (Tensor) –1-D Tensor of shape (M), which equals to *shape[0] + 1*, which indicates the start and end point for *values* in each row. Default: None. If provided, must be int16, int32 or int64.
- **indices** (Tensor) –1-D Tensor of shape (N), which has the same length as *values*. *indices* indicates the which column *values* should be placed. Default: None. If provided, must be int16, int32 or int64.
- **values** (Tensor) –Tensor, which has the same length as *indices* (values.shape[0] == indices.shape[0]). *values* stores the data for CSRTensor. Default: None.
- **shape** (*tuple(int)*) –An integer tuple of shape $(ndims)$, and *shape[0]* must equal to *M - 1*, which all equal to number of rows of the CSRTensor. Default: None.
- **csr_tensor** (CSRTensor) –A CSRTensor object. Values' feature dimension should match with CSRTensor's feature dimension ($values.shape[1:] == csr\_tensor.shape[2:]$) . Default: None.

Outputs:

CSRTensor, with shape defined by *shape*, and dtype inferred from *value*.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> # initialize a csr_tensor with indptr, indices, values and shape
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> # access a data member of CSRTensor
>>> print(indptr == csr_tensor.indptr)
[ True  True  True]
```

abs()

Return absolute value element-wisely.

Returns

CSRTensor, with all values being non-negative.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([-1, -2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.abs().values)
[1. 2.]
```

add(*b:* CSRTensor, *alpha:* Tensor, *beta:* Tensor)

Addition of two CSR Tensors : C = alpha * A + beta * B

Parameters

- **b** (CSRTensor) –Sparse CSR Tensor.
- **alpha** (Tensor) –Dense Tensor, its shape must be able to broadcast to self.
- **beta** (Tensor) –Dense Tensor, its shape must be able to broadcast to b.

Returns

CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> import mindspore.common.dtype as mstype
>>> indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> indices = Tensor([0, 1], dtype=mstype.int32)
>>> values_a = Tensor([2, 1], dtype=mstype.float32)
>>> values_b = Tensor([1, 2], dtype=mstype.float32)
>>> dense_shape = (2, 4)
>>> alpha = Tensor(1, mstype.float32)
>>> beta = Tensor(1, mstype.float32)
>>> a = CSRTensor(indptr, indices, values_a, dense_shape)
>>> b = CSRTensor(indptr, indices, values_b, dense_shape)
>>> print(a.add(b, alpha, beta))
CSRTensor(shape=[2, 4], dtype=Float32,
          indptr=Tensor(shape=[3], dtype=Int32, value=[0 1 2]),
          indices=Tensor(shape=[2], dtype=Int32, value=[0 1]),
          values=Tensor(shape=[2], dtype=Float32, value=[ 3.00000000e+00  3.
↪00000000e+00]))
```

astype(*dtype: mstype*)

Return a copy of the CSRTensor, cast its values to a specified type.

Parameters

dtype (Union[*mindspore.dtype*, numpy.dtype, str]) –Designated tensor dtype.

Returns

CSRTensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.astype(ms.float64).dtype)
Float64
```

property dtype: mstype

Return the dtype of the values of CSRTensor (*mindspore.dtype*).

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.dtype)
Float32
```

property indices: mindspore.common.tensor.Tensor

Return CSRTensor's column indices.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.indices)
[0 1]
```

property indptr: mindspore.common.tensor.Tensor

Return CSRTensor's row indices pointers.

property itemsize: int

Return the length of one tensor element in bytes.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float64)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.itemsize)
8
```

mm(*matrix: Union[*Tensor, CSRTensor*]*)

Return the matrix multiplication result of the right-multiply matrix(dense or CSRTensor) of the CSRTensor. The CSRTensor with shape *[M, N]* needs to adapt the right matrix with shape *[N, K]* to get the dense matrix or CSRTensor with result *[M, K]*.

Note: If right matrix is CSRTensor, currently only supports GPU backend. If right matrix is Tensor, currently supports CPU backend with LLVM no lower than 12.0.1, and GPU backend.

Parameters

matrix (Tensor *or* CSRTensor) – A dense Tensor or CSRTensor, its shape[0] should be equal to csr_tensor.shape[1]

Returns

Tensor or CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> from mindspore import dtype as mstype
>>> indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> indices = Tensor([0, 1], dtype=mstype.int32)
>>> values = Tensor([2, 1], dtype=mstype.float32)
>>> dense_shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, dense_shape)
>>> dense_matrix = Tensor([[1., 2.], [1, 2.], [1, 2.], [1., 2.]], dtype=mstype.float32)
>>> print(csr_tensor.mm(dense_matrix))
[[2. 4.]
[1. 2.]]
```

mv(*dense_vector:* Tensor)

Return the matrix multiplication result of the right-multiply dense matrix of the CSRTensor. The CSRTensor with shape *[M, N]* needs to adapt the dense vector with shape *[N, 1]* to get the dense vector with result *[M, 1]*.

Note: Currently only supports CPU backend with LLVM 12.0.1 installed.

Parameters

dense_vector (Tensor) −A dense Tensor, its shape must be (csr_tensor.shape[1], 1)

Returns

Tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> from mindspore import dtype as mstype
>>> indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> indices = Tensor([0, 1], dtype=mstype.int32)
>>> values = Tensor([2, 1], dtype=mstype.float32)
>>> dense_shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, dense_shape)
>>> dense = Tensor([[1], [1], [1], [1]], dtype=mstype.float32)
>>> print(csr_tensor.mv(dense))
[[2.]
[1.]]
```

property ndim: int

Return the number of tensor dimensions.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.ndim)
2
```

property shape: Tuple[int, ...]

Return CSRTensor's shape.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.shape)
(2, 4)
```

property size: int
Return the number of non-zero values.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.size)
2
```

sum(*axis: int*)
Reduces a dimension of a CSRTensor by summing all elements in the dimension.

Note: Currently only supports CPU backend with LLVM 12.0.1 installed.

Parameters

axis (*int*) – The dimensions to reduce.

Returns

Tensor, the dtype is the same as *CSRTensor.values*.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> from mindspore import dtype as mstype
>>> indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> indices = Tensor([0, 1], dtype=mstype.int32)
>>> values = Tensor([2, 1], dtype=mstype.float32)
>>> dense_shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, dense_shape)
>>> print(csr_tensor.sum(1))
[[2.]
[1.]]
```

to_coo()
Converts CSRTensor to COOTensor.

Note: Currently only supports CPU backend with LLVM 12.0.1 installed.

Returns

COOTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.int32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.to_coo())
COOTensor(shape=[2, 4], dtype=Int32, indices=Tensor(shape=[2, 2], dtype=Int32, value=
[[0 0]
 [1 1]]), values=Tensor(shape=[2], dtype=Int32, value=[1 2]))
```

to_dense()

Converts CSRTensor to Dense Tensor.

Returns

Tensor.

Supported Platforms:

GPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.to_dense())
[[1. 0. 0. 0.]
 [0. 2. 0. 0.]]
```

to_tuple()

Return indptr, indices, values and shape as a tuple.

Returns

Tuple.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.to_tuple())
(Tensor(shape=[3], dtype=Int32, value= [0, 1, 2]), Tensor(shape=[2], dtype=Int32, value=␣
↪[0, 1]),
    Tensor(shape=[2], dtype=Float32, value= [ 1.00000000e+00,  2.00000000e+00]), (2, 4))
```

property values: mindspore.common.tensor.Tensor

Return CSRTensor's non-zero values.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2], dtype=ms.int32)
>>> indices = Tensor([0, 1], dtype=ms.int32)
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (2, 4)
>>> csr_tensor = CSRTensor(indptr, indices, values, shape)
>>> print(csr_tensor.values)
[1. 2.]
```

mindspore.RowTensor

class mindspore.**RowTensor**(*indices=None*, *values=None*, *shape=None*, *row_tensor=None*)

A sparse representation of a set of tensor slices at given indices.

When the *values* of a RowTensor has a shape of $(d_0, d_1, ..., d_n)$, then this RowTensor is used to represent a subset of a larger dense tensor of shape $(l_0, d_1, ..., d_n)$, where d_i is the size of i-th axis in RowTensor, l_0 is the size of 0-th axis of dense tensor and it satisfies $l_0 > d_0$.

The parameter *indices* is used to specify locations from which the *RowTensor* is sliced in the first dimension of the dense tensor, which means the parameters *indices* and *values* have the following relationship $dense[indices[i], :, :, ...] = values[i, :, :, ...]$.

For example, if indices is [0], values is [[1, 2]], shape is $(3, 2)$, then the dense representation of the row tensor will be:

```
[[1, 2],
 [0, 0],
 [0, 0]]
```

Warning:

- This is an experimental API that is subjected to change or deletion.
- If use PyNative mode, set "export MS_PYNATIVE_CONFIG_STATIC_SHAPE=1".

Parameters

- **indices** (Tensor) −A 1-D integer Tensor of shape (d_0) . Default: None.
- **values** (Tensor) −A Tensor of any dtype of shape $(d_0, d_1, ..., d_n)$. Default: None.
- **shape** (*tuple(int)*) −An integer tuple which contains the shape of the corresponding dense tensor. Default: None.
- **row_tensor** (RowTensor) −A RowTensor object. Default: None.

Returns

RowTensor, composed of *indices*, *values*, and *shape*.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, RowTensor
>>> indices = Tensor([0])
>>> values = Tensor([[1, 2]], dtype=ms.float32)
>>> shape = (3, 2)
>>> x = RowTensor(indices, values, shape)
>>> print(x.values)
[[1. 2.]]
>>> print(x.indices)
[0]
>>> print(x.dense_shape)
(3, 2)
```

mindspore.SparseTensor

class mindspore.**SparseTensor**(*indices*, *values*, *shape*)

A sparse representation of a set of nonzero elements from a tensor at given indices.

SparseTensor can only be used in the *Cell*'s construct method.

For a tensor dense, its SparseTensor(indices, values, dense_shape) has *dense[indices[i]] = values[i]*.

For example, if indices is [[0, 1], [1, 2]], values is [1, 2], dense_shape is (3, 4), then the dense representation of the sparse tensor will be:

```
[[0, 1, 0, 0],
 [0, 0, 2, 0],
 [0, 0, 0, 0]]
```

Note: The interface is deprecated from version 1.7 and will be removed in a future version. Please use *mindspore.COOTensor* instead.

Parameters

- **indices** (Tensor) −A 2-D integer Tensor of shape $(N, ndims)$, where N and ndims are the number of *values* and number of dimensions in the SparseTensor, respectively.
- **values** (Tensor) −A 1-D tensor of any type and shape (N), which supplies the values for each element in *indices*.
- **shape** (*tuple(int)*) −An integer tuple of size $(ndims)$, which specifies the shape of the sparse tensor.

Returns

SparseTensor, composed of *indices*, *values*, and *shape*.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, SparseTensor
>>> indices = Tensor([[0, 1], [1, 2]])
>>> values = Tensor([1, 2], dtype=ms.float32)
>>> shape = (3, 4)
>>> x = SparseTensor(indices, values, shape)
>>> print(x.values)
[1. 2.]
>>> print(x.indices)
[[0 1]
 [1 2]]
>>> print(x.shape)
(3, 4)
```

property indices
Return SparseTensor's indices.

property shape
Return SparseTensor's shape.

property values
Return SparseTensor's non-zero values.

mindspore.is_tensor

mindspore.**is_tensor**(*obj*)
Check whether the input object is *mindspore.Tensor*.

Parameters

obj (*Object*) – input object.

Returns

Bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([1.9, 2.2, 3.1])
>>> mindspore.ops.is_tensor(a)
True
```

mindspore.from_numpy

mindspore.**from_numpy**(*array*)

Convert numpy array to Tensor. If the data is not C contiguous, the data will be copied to C contiguous, then construct the tensor. Otherwise, the tensor will be constructed using this numpy array without copy.

Parameters

array (*numpy.array*) – The input array.

Returns

Tensor, has the same data type as input array.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.array([1, 2])
>>> output = ms.from_numpy(x)
>>> print(output)
[1 2]
```

1.1.2 Parameter

mindspore.Parameter	*Parameter* is a *Tensor* subclass, when they are assigned as Cell attributes they are automatically added to the list of its parameters, and will appear, e.g.
mindspore.ParameterTuple	Inherited from tuple, ParameterTuple is used to save multiple parameter.

mindspore.Parameter

class mindspore.**Parameter**(*default_input*, *name=None*, *requires_grad=True*, *layerwise_parallel=False*, *parallel_optimizer=True*, *storage_format=''*, *device=None*)

Parameter is a *Tensor* subclass, when they are assigned as Cell attributes they are automatically added to the list of its parameters, and will appear, e.g. in *cell.get_parameters()* iterator.

Note:

- When using *AutoParallel(cell)* to enable parallel mode, if init *Parameter* by a *Tensor*, the type of Parameter will be *Tensor*. *Tensor* will save the shape and type info of a tensor with no memory usage.
- The shape can be changed while compiling for auto-parallel. Call *init_data* will return a Tensor Parameter with initialized data.
- If there is an operator in the network that requires part of the inputs to be Parameter, then the Parameters as this part of the inputs are not allowed to be cast.
- Give each *Parameter* a unique name to facilitate subsequent operations and updates. If there are two or more *Parameter* objects with the same name in a network, will be prompted to set a unique name when defining.
- When directly printing a *Parameter*, you cannot view the actual values contained inside it. You need to use the *Parameter.asnumpy()* method to access the actual values.

Parameters

- **default_input** (*Union[Tensor, int, float, numpy.ndarray, list]*) –Parameter data, to initialize the parameter data.
- **name** (*str*) –Name of the parameter. Default: `None` . If two or more *Parameter* objects with the same name exist in a network, you will be prompted to set a unique name when defining them.

  1) If the parameter is not given a name, the default name is its variable name. For example, the name of param_a below is name_a, and the name of param_b is the variable name param_b.

```
self.param_a = Parameter(Tensor([1], ms.float32), name="name_a")
self.param_b = Parameter(Tensor([2], ms.float32))
```

  2) If parameter in list or tuple is not given a name, will give it a unique name. For example, the names of parameters below are **Parameter$1** and **Parameter$2**.

```
self.param_list = [Parameter(Tensor([3], ms.float32)),
                   Parameter(Tensor([4], ms.float32))]
```

  3) If the parameter is given a name, and the same name exists between different parameters, an exception will be thrown. For example, "its name 'name_a' already exists." will be thrown.

```
self.param_a = Parameter(Tensor([1], ms.float32), name="name_a")
self.param_tuple = (Parameter(Tensor([5], ms.float32), name="name_a"),
                    Parameter(Tensor([6], ms.float32)))
```

  4) If a parameter appear multiple times in list or tuple, check the name of the object only once. For example, the following example will not throw an exception.

```
self.param_a = Parameter(Tensor([1], ms.float32), name="name_a")
self.param_tuple = (self.param_a, self.param_a)
```

- **requires_grad** (*bool*) –True if the parameter requires gradient. Default: `True` .
- **layerwise_parallel** (*bool*) –When *layerwise_parallel* is true in data/hybrid parallel mode, broadcast and gradients communication would not be applied to the *Parameter*. Default: `False` .
- **parallel_optimizer** (*bool*) –It is used to filter the weight shard operation in parallel mode. It works only when enable parallel optimizer in *mindspore.parallel.auto_parallel.AutoParallel.hsdp()*. Default: `True` .
- **storage_format** (*str*) –Only Ascend device target is supported. It is used to specify the format of the weight loaded to the device. By default, the format is not changed. The optional values are `"FRACTAL_NZ"` , `"NC1HWC0"` , `"FRACTAL_Z"` , etc. Default: `""` .
- **device** (*str*) –Only Ascend device target is supported. It is used to specify the device which the parameter is stored. By default, the parameter will be stored on NPU while computing. When the device is specified as `"CPU"`, the parameter will be loaded into the device when it needs to be used, and unloaded to the CPU after use. It takes effext only when *memory_offload* is `"ON"`, *jit_level* is not `"O2"` and *memory_optimize_level* is `O0` in *mindspore.set_context()*. Less device memory is needed when device is specified as `"CPU"`.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Parameter, Tensor, ops, nn
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.matmul = ops.MatMul()
...         self.weight = Parameter(Tensor(np.ones((1, 2)), mindspore.float32), name="w",␣
↪requires_grad=True)
...
...     def construct(self, x):
...         out = self.matmul(self.weight, x)
...         return out
>>> net = Net()
>>> x = Tensor(np.ones((2, 1)), mindspore.float32)
>>> print(net(x))
[[2.]]
>>> net.weight.set_data(Tensor(np.zeros((1, 2)), mindspore.float32))
>>> print(net(x))
[[0.]]
```

add_pipeline_stage(*stage*)

Add a pipeline stage to the parameter.

Parameters

stage (*int*) – The pipeline stage to be added.

Raises

TypeError – If *stage* is not a positive number or not int type.

property cache_enable

Return whether the parameter is cache enable.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.cache_enable=True
>>> x.cache_enable
True
```

property cache_shape

Return the cache shape corresponding to the parameter if use cache.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.cache_enable=True
>>> x.cache_shape=[1, 2]
>>> x.cache_shape
[1, 2]
```

clone(*init='same'*)

Clone the parameter.

Parameters

init (*Union[Tensor, str, numbers.Number]*) – Initialize the shape and dtype of the parameter. If *init* is a *Tensor* or *numbers.Number*, clone a new parameter with the same shape and dtype, and the data of the new parameter will be set according to *init*. If *init* is a *str*, the *init* should be the alias of the class inheriting from *Initializer*. For example, if *init* is `'same'`, clone a new parameter with the same data, shape, and dtype. Default: `'same'`.

Returns

Parameter, a new parameter.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> y = x.clone()
```

property comm_fusion

Get the fusion type (int) for communication operators corresponding to this parameter.

When using *AutoParallel(cell)* to enable parallel mode, some communication operators used for parameters or gradients aggregation are inserted automatically. The value of *comm_fusion* must be greater than or equal to 0. When the value of *comm_fusion* is `0` , operators will not be fused together.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.comm_fusion = 3
>>> x.comm_fusion
3
```

copy()

Copy the parameter.

Returns

Parameter, a new parameter.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> y = x.copy()
```

property data

Return the parameter object.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([[1, 2], [3, 4]], dtype=np.float32)), name="param")
>>> x.data
Parameter (name=param, shape=(2, 2), dtype=Float32, requires_grad=True)
```

init_data(*layout=None*, *set_sliced=False*)

Initialize the parameter's data.

Parameters

- **layout** (*Union[None, tuple]*) – The parameter's layout info. layout [dev_mat, tensor_map, slice_shape, filed_size, uniform_split, opt_shard_group]. Default: `None`. It's not None only when using *AutoParallel(cell)* to enable parallel mode.
  - dev_mat (list(int)): The parameter's device matrix.
  - tensor_map (list(int)): The parameter's tensor map.
  - slice_shape (list(int)): The parameter's slice shape.
  - filed_size (int): The parameter's filed size.
  - uniform_split (bool): Whether the parameter is split evenly.
  - opt_shard_group (str): The group of the parameter while running optimizer parallel.
- **set_sliced** (*bool*) – True if the parameter is set sliced after initializing the data. Default: `False`.

Returns

Parameter, the *Parameter* after initializing data. If current *Parameter* was already initialized before, returns the same initialized *Parameter*.

Raises

- **RuntimeError** – If it is from Initializer, and parallel mode has changed after the Initializer created.
- **ValueError** – If the length of the layout is less than 6.
- **TypeError** – If *layout* is not tuple.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([[1, 2], [3, 4]], dtype=np.float32)), name="param")
>>> x.init_data()
```

property inited_param

Get the new parameter after call the init_data.

Default is a None, If *self* is a Parameter without data, after call the *init_data* the initialized Parameter with data will be recorded here.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.inited_param
```

property key

Return the parameter unique key.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.key = 2
>>> x.key
2
```

property layerwise_parallel

Get the layerwise parallel status(bool) of the parameter.

When *layerwise_parallel* is `True` in *DATA_PARALLEL* and *HYBRID_PARALLEL* parallel mode, broadcast and gradients communication would not be applied to parameters.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.layerwise_parallel = True
>>> x.layerwise_parallel
True
```

property name

Get the name of the parameter.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.name = "param1"
>>> x.name
'param1'
```

property parallel_optimizer

Get the optimizer parallel status(bool) of the parameter.

When using *AutoParallel(cell)* to enable parallel mode, it is used to filter the weight shard operation. It works only when enable parallel optimizer in *mindspore.parallel.auto_parallel.AutoParallel.hsdp()*.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.parallel_optimizer = True
>>> x.parallel_optimizer
True
```

property parallel_optimizer_comm_recompute

Get the communication recompute status(bool) of optimizer parallel for the parameter.

When using *AutoParallel(cell)* to enable parallel mode, and applying parallel optimizer, some *mindspore.ops.AllGather* operators used for parameters gathering are inserted automatically. It is used to control the recompute attr for those *mindspore.ops.AllGather* operators.

Note:

- Only *Graph* mode is supported.
- It is recommended to use cell.recompute(parallel_optimizer_comm_recompute=True/False) to configure the AllGather operators introducing by parallel optimizer rather than using this interface directly.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.parallel_optimizer_comm_recompute = True
>>> x.parallel_optimizer_comm_recompute
True
```

register_hook(*hook_fn*)

For details, please refer to *mindspore.Tensor.register_hook()*.

property requires_grad

Return whether the parameter requires gradient.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.requires_grad = True
>>> x.requires_grad
True
```

set_data(*data*, *slice_shape=False*)

Set Parameter's data.

Parameters

- **data** (*Union[Tensor, int, float]*) – New data.
- **slice_shape** (*bool*) – If slice the parameter is set to `True`, the shape consistency will not be checked. Default: `False`. When *slice_shape* is `True`, and the shapes are not consistent, a ValueError will be thrown.

Returns

Parameter, the parameter after set data.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([[1, 2], [3, 4]], dtype=np.float32)), name="param")
>>> x.set_data(Tensor(np.array([[6, 6], [6, 6]], dtype=np.float32)))
Parameter (name=param, shape=(2, 2), dtype=Float32, requires_grad=True)
```

set_param_ps(*init_in_server=False*)

Set whether the trainable parameter is updated by parameter server and whether the trainable parameter is initialized on server.

Note: It only works when a running task is in the parameter server mode. It is supported only in graph mode.

Parameters

init_in_server (*bool*) – Whether trainable parameter updated by parameter server is initialized on server. Default: `False`.

property sliced

Get slice status of the parameter.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.sliced = True
>>> x.sliced
True
```

property unique
Whether the parameter is already unique or not.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x.unique = True
>>> x.unique
True
```

value()
Return the value of parameter object.

Examples

```
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([1, 2], dtype=np.float32)), name="param")
>>> x_value = x.value()
>>> print(x_value)
[1.  2.]
```

mindspore.ParameterTuple

class mindspore.**ParameterTuple**
Inherited from tuple, ParameterTuple is used to save multiple parameter.

Note: It is used to store the parameters of the network into the parameter tuple collection.

Examples

```
>>> from mindspore import Tensor, Parameter, ParameterTuple
>>> import numpy as np
>>> x = Parameter(Tensor(np.array([[1, 2], [3, 4]], dtype=np.float32)), name="param")
>>> y = Parameter(Tensor(np.array([[5, 6], [7, 8]], dtype=np.float32)), name="param1")
>>> pt = ParameterTuple([x, y])
>>> pt1 = pt.clone(prefix="new")
```

clone(*prefix*, *init='same'*)
Clone the parameters in ParameterTuple element-wisely to generate a new ParameterTuple.

Parameters

- **prefix** (*str*) –Namespace of parameter, the prefix string will be added to the names of parameters in parametertuple.
- **init** (*Union[Tensor, str, numbers.Number]*) –Clone the shape and dtype of Parameters in ParameterTuple and set data according to *init*. Default: `'same'`.
  - If *init* is a *Tensor* , set the new Parameter data to the input Tensor.
  - If *init* is a *str*, data will be set according to the initialization method of the same name in the *Initializer*. When it is `'same'`, the new Parameter will have the same value with the original Parameter.
  - If *init* is *numbers.Number* , set the new Parameter data to the input number.

Returns

Tuple, the new Parameter tuple.

1.1.3 DataType

mindspore.dtype	Data type for MindSpore.
mindspore.dtype_to_nptype	Convert MindSpore dtype to numpy data type.
mindspore.dtype_to_pytype	Convert MindSpore dtype to python data type.
mindspore.pytype_to_dtype	Convert python type to MindSpore type.
mindspore.get_py_obj_dtype	Get the MindSpore data type, which corresponds to python type or variable.
mindspore.QuantDtype	An enum for quant datatype, contains *INT1* ~ *INT16*, *UINT1* ~ *UINT16*.
mindspore.common.np_dtype	Numpy data type for MindSpore.

mindspore.dtype

class mindspore.**dtype**
Data type for MindSpore.

The actual path of `dtype` is `/mindspore/common/dtype.py`. Run the following command to import the package:

```
from mindspore import dtype as mstype
```

Basic Data Type

MindSpore supports the following base data types:

Definition	Description
`mindspore.int8` , `mindspore.byte`	8-bit integer
`mindspore.int16` , `mindspore.short`	16-bit integer
`mindspore.int32` , `mindspore.intc`	32-bit integer
`mindspore.int64` , `mindspore.intp`	64-bit integer
`mindspore.uint8` , `mindspore.ubyte`	unsigned 8-bit integer
`mindspore.uint16` , `mindspore.ushort`	unsigned 16-bit integer
`mindspore.uint32` , `mindspore.uintc`	unsigned 32-bit integer
`mindspore.uint64` , `mindspore.uintp`	unsigned 64-bit integer
`mindspore.float16` , `mindspore.half`	16-bit floating-point number
`mindspore.float32` , `mindspore.single`	32-bit floating-point number
`mindspore.float64` , `mindspore.double`	64-bit floating-point number
`mindspore.bfloat16`	16-bit brain-floating-point number
`mindspore.complex64`	64-bit complex number
`mindspore.complex128`	128-bit complex number

Other Type

For other defined types, see the following table.

Type	Description
`tensor`	MindSpore's `tensor` type. Data format uses NCHW. For details, see tensor.
`bool_`	Boolean `True` or `False`.
`int_`	Integer scalar.
`uint`	Unsigned integer scalar.
`float_`	Floating-point scalar.
`complex`	Complex scalar.
`number`	Number, including `int_` , `uint` , `float_` , `complex` and `bool_` .
`list_`	List constructed by `tensor` , such as `List[T0,T1,...,Tn]` , where the element `Ti` can be of different types.
`tuple_`	Tuple constructed by `tensor` , such as `Tuple[T0,T1,...,Tn]` , where the element `Ti` can be of different types.
`function`	Function. Return in two ways, when function is not None, returns Func directly, the other returns Func(args: List[T0,T1,⋯,Tn], retval: T) when function is None.
`type_type`	Type definition of type.
`type_none`	No matching return type, corresponding to the `type(None)` in Python.
`symbolic_key`	The value of a variable is used as a key of the variable in `env_type` .
`env_type`	Used to store the gradient of the free variable of a function, where the key is the `symbolic_key` of the free variable's node and the value is the gradient.

Type conversion rules

When some inputs of an operator are required to have the same target type, type promotion will be automatically performed according to the type conversion rules. If these inputs have types of different sizes and categories (where `complex > float > int > bool`), they will be promoted a type with sufficient size and category.

For the type conversion rules between Tensor and Tensor, please refer to the following table. The first row and the first column in the table both represent the types of the input `Tensor`, and the corresponding position in the table represents the type of the output `Tensor`. – indicates that no type promotion will be performed.

For convenience of description, `bool_` is used in the table to refer to `mindspore.bool_`, `int8` is used to refer to `mindspore.int8`, and so on.

Tensor and Tensor	**bool_**	**int8**	**int16**	**int32**	**int64**	**uint8**	**uint16**	**uint32**	**uint64**	**float16**	**bfloat16**	**float32**	**float64**	**complex64**	**complex128**
bool_	bool_	int8	int16	int32	int64	uint8	uint16	uint32	uint64	float16	bfloat16	float32	float64	complex64	complex128
int8	int8	int8	int16	int32	int64	int16	–	–	–	float16	bfloat16	float32	float64	complex64	complex128
int16	int16	int16	int16	int32	int64	int16	–	–	–	float16	bfloat16	float32	float64	complex64	complex128
int32	int32	int32	int32	int32	int64	int32	–	–	–	float16	bfloat16	float32	float64	complex64	complex128
int64	int64	int64	int64	int64	int64	int64	–	–	–	float16	bfloat16	float32	float64	complex64	complex128
uint8	uint8	int16	int16	int32	int64	uint8	–	–	–	float16	bfloat16	float32	float64	complex64	complex128
uint16	uint16	–	–	–	–	–	uint16	–	–	–	–	–	–	–	–
uint32	uint32	–	–	–	–	–	–	uint32	–	–	–	–	–	–	–
uint64	uint64	–	–	–	–	–	–	–	uint64	–	–	–	–	–	–
float16	float16	float16	float16	float16	float16	float16	float16	–	–	float16	float32	float32	float64	complex64	complex128
bfloat16	bfloat16	bfloat16	bfloat16	bfloat16	bfloat16	bfloat16	bfloat16	–	–	float32	bfloat16	float32	float64	complex64	complex128
float32	float32	float32	float32	float32	float32	float32	float32	–	–	float32	float32	float32	float64	complex64	complex128
float64	float64	float64	float64	float64	float64	float64	float64	–	–	float64	float64	float64	float64	complex128	complex128
complex64	complex64	complex64	complex64	complex64	complex64	complex64	–	–	–	complex64	complex64	complex64	complex128	complex64	complex128
complex128	complex128	complex128	complex128	complex128	complex128	complex128	–	–	–	complex128	complex128	complex128	complex128	complex128	complex128

For the type conversion rules between Number and Tensor, please refer to the following table. The first row in the table indicates the type of the input `Number`, and the first column indicates the types of input `Tensor`. The corresponding position in the table represents the type of the output `Tensor`. – indicates that no type promotion will be performed.

Number and Tensor	**bool**	**int**	**float**
bool_	bool_	int64	float32
int8	int8	int8	float32
int16	int16	int16	float32
int32	int32	int32	float32
int64	int64	int64	float32
uint8	uint8	uint8	float32
uint16	uint16	-	-
uint32	uint32	-	-
uint64	uint64	-	-
float16	float16	float16	float16
bfloat16	bfloat16	bfloat16	bfloat16
float32	float32	float32	float32
float64	float64	float64	float64
complex64	complex64	complex64	complex64
complex128	complex128	complex128	complex128

mindspore.dtype_to_nptype

mindspore.**dtype_to_nptype**(*type_*)

Convert MindSpore dtype to numpy data type.

Parameters

type_ (*mindspore.dtype*) – MindSpore's dtype.

Returns

The data type of numpy.

Examples

```
>>> import mindspore as ms
>>> ms.dtype_to_nptype(ms.int8)
<class 'numpy.int8'>
```

mindspore.dtype_to_pytype

mindspore.**dtype_to_pytype**(*type_*)

Convert MindSpore dtype to python data type.

Parameters

type_ (*mindspore.dtype*) – MindSpore's dtype.

Returns

Type of python.

Examples

```
>>> import mindspore as ms
>>> out = ms.dtype_to_pytype(ms.bool_)
>>> print(out)
<class 'bool'>
```

mindspore.pytype_to_dtype

mindspore.**pytype_to_dtype**(*obj*)

Convert python type to MindSpore type.

Parameters

obj (*type*) – A python type object.

Returns

Type of MindSpore type.

Raises

NotImplementedError – If the python type cannot be converted to MindSpore type.

Examples

```
>>> import mindspore as ms
>>> out = ms.pytype_to_dtype(bool)
>>> print(out)
Bool
```

mindspore.get_py_obj_dtype

mindspore.**get_py_obj_dtype**(*obj*)

Get the MindSpore data type, which corresponds to python type or variable.

Parameters

obj (*type*) – An object of python type, or a variable of python type.

Returns

Type of MindSpore type.

Examples

```
>>> import mindspore as ms
>>> ms.get_py_obj_dtype(1)
mindspore.int64
```

mindspore.QuantDtype

class mindspore.**QuantDtype**

An enum for quant datatype, contains *INT1 ~ INT16*, *UINT1 ~ UINT16*.

QuantDtype is defined in dtype.py , use command below to import:

```
from mindspore import QuantDtype
```

Tutorial Examples:

- Quantization algorithm in Golden Stick

value()

Return value of *QuantDtype*. This interface is currently used to serialize or deserialize *QuantDtype* primarily.

Returns

An int as value of *QuantDtype*.

Examples

```
>>> from mindspore import QuantDtype
>>> print(QuantDtype.INT8.value())
7
>>> print(QuantDtype.UINT16.value())
115
```

mindspore.common.np_dtype

class mindspore.common.**np_dtype**

Numpy data type for MindSpore.

The actual path of `np_dtype` is `/mindspore/common/np_dtype.py`. Run the following command to import the package:

```
from mindspore.common import np_dtype
```

- **Numeric Type**

Type	Description
`bfloat16`	The `bfloat16` data type under NumPy. This type is only used to construct Tensor of type `bfloat16`, and does not guarantee the full computing power under Numpy. Takes effect only if the version of Numpy at runtime is not less than the version of Numpy at compilation.

1.2 Context

mindspore.set_device	Set device target and device id for running environment.
mindspore.get_current_device	Get device target and device id in the current running environment.
mindspore.set_deterministic	Enables or disables deterministic computing.
mindspore.set_context	Set context for running environment, this interface will be deprecated in future versions, and its parameter-related functionalities will be provided through new APIs.
mindspore.get_context	Get context attribute value according to the input key, this api will be deprecated and removed in future versions, please use *mindspore.get_current_device()* instead.
mindspore.set _auto_parallel_context	Set auto parallel context, this api will be deprecated and removed in future versions, please use the api *mindspore.parallel.auto_parallel.AutoParallel* instead.
mindspore.get _auto_parallel_context	Get auto parallel context attribute value according to the key, this api will be deprecated and removed in future versions.
mindspore.reset _auto_parallel_context	Reset auto parallel context attributes to the default values, this api will be deprecated and removed in future versions, please use the api *mindspore.parallel.auto_parallel.AutoParallel* instead.
mindspore.ParallelMode	Parallel mode options.
mindspore.set_ps_context	Set parameter server training mode context, this api will be deprecated and removed in future versions.
mindspore.get_ps_context	Get parameter server training mode context attribute value according to the key, this api will be deprecated and removed in future versions.
mindspore.reset_ps_context	Reset parameter server training mode context attributes to the default values, this api will be deprecated and removed in future versions.
mindspore.set_algo_parameters	Set parameters in the algorithm for parallel strategy searching.
mindspore.get_algo_parameters	Get the algorithm parameter config attributes.
mindspore.reset_algo_parameters	Reset the algorithm parameter attributes.
mindspore.set_offload_context	Configure heterogeneous training detailed parameters to adjust the offload strategy, this api will be deprecated and removed in future versions.
mindspore.get_offload_context	Gets the offload configuration parameters, this api will be deprecated and removed in future versions.

1.2.1 mindspore.set_device

mindspore.**set_device**(*device_target*, *device_id=None*)

Set device target and device id for running environment.

Note:

- The *device_target* must be set in the ["CPU", "GPU", "Ascend"], there is no default value.
- Suggest setting *device_target* and *device_id* before calling *mindspore.communication.init()*.

Parameters

- **device_target** (*str*) –The target device to run, only support "Ascend", "GPU", and "CPU".
- **device_id** (*int, optional*) –ID of the target device, the value must be in [0, device_num_per_host-1], where device_num_per_host refers to the total number of devices on the host. Default: `None` . The frame will set different default behaviours according to the scenario: if it is a single-card scenario, the frame will be set to 0. In a distributed scenario where msrun is started, the framework will automatically negotiate the available device_id values. In a distributed scenario with other startup methods, the frame is set to 0.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
```

1.2.2 mindspore.get_current_device

mindspore.**get_current_device**()

Get device target and device id in the current running environment.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.get_current_device()
('Ascend', 1)
>>> ms.get_current_device().device_target
'Ascend'
>>> ms.get_current_device().device_id
1
```

1.2.3 mindspore.set_deterministic

mindspore.**set_deterministic**(*deterministic*)

Enables or disables deterministic computing.

When deterministic computing is enabled, the same output is generated if an operator is executed for multiple times with the same hardware and input.This often slows down operator execution. In distributed scenario, we suggest user to set deterministic mode before calling *mindspore.communication.init()* to enable deterministic operation for communication operators in the global communication group.

The framework not enabled deterministic computation by default.

Parameters

deterministic (*bool*) –Whether to enable deterministic computing.

Examples

```
>>> import mindspore as ms
>>> ms.set_deterministic(True)
```

1.2.4 mindspore.set_context

mindspore.**set_context**(***kwargs*)

Set context for running environment, this interface will be deprecated in future versions, and its parameter-related functionalities will be provided through new APIs.

Parameters

- **mode** (*int*) –GRAPH_MODE(0) or PYNATIVE_MODE(1). Default `PYNATIVE_MODE` .
- **device_id** (*int*) –ID of the target device. Default `0` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.set_device()* instead.
- **device_target** (*str*) –The target device to run, support `"Ascend"`, `"GPU"`, and `"CPU"`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.set_device()* instead.
- **deterministic** (*str*) –Deterministic computation of operators. Default `"OFF"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.set_deterministic()* instead.
- **max_call_depth** (*int*) –The maximum depth of function call. Default `1000` . This parameter will be deprecated and removed in a future version. Please use the api *mindspore.set_recursion_limit()* instead.
- **variable_memory_max_size** (*str*) –This parameter will be deprecated and removed in future versions. Please use the api *mindspore.runtime.set_memory()* instead.
- **mempool_block_size** (*str*) –Set the size of the memory pool block for devices. Default `"1GB"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.runtime.set_memory()* instead.
- **memory_optimize_level** (*str*) –The memory optimize level. Default `"O0"`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.runtime.set_memory()* instead.
- **max_device_memory** (*str*) –Set the maximum memory available for devices. Default `"1024GB"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.runtime.set_memory()* instead.
- **pynative_synchronize** (*bool*) –Whether to enable synchronous execution of the device in PyNative mode. Default `False` . This parameter will be deprecated and removed in future versions.Please use the api *mindspore.runtime.launch_blocking()* instead.
- **compile_cache_path** (*str*) –Path to save the compile cache. Default `"."`. This parameter will be deprecated and removed in a future version. Please use the environment variable *MS_COMPILER_CACHE_PATH* instead.
- **inter_op_parallel_num** (*int*) –The thread number of op parallel at the same time. Default `0` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.runtime.dispatch_threads_num()* instead.
- **memory_offload** (*str*) –Whether to enable the memory offload function. Default `"OFF"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.nn.Cell.offload()* instead.
- **disable_format_transform** (*bool*) –Whether to disable the automatic format transform function from NCHW to NHWC. Default `False` . This parameter will be deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.

- **jit_syntax_level** (*int*) −Set JIT syntax support level. Default `LAX` . This parameter is deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
- **jit_config** (*dict*) −Set the global jit config for compile. This parameter is deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
- **exec_order** (*str*) −The sorting method for operator execution. This parameter is deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
- **op_timeout** (*int*) −Set the maximum duration of executing an operator in seconds. Default `900` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_debug.execute_timeout()* instead.
- **aoe_tune_mode** (*str*) −AOE tuning mode. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_tuning.aoe_tune_mode()* instead.
- **aoe_config** (*dict*) −AOE-specific parameters. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_tuning.aoe_job_type()* instead.
- **runtime_num_threads** (*int*) −The thread pool number of cpu kernel used in runtime. Default `30` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.cpu.op_tuning.threads_num()* instead.
- **save_graphs** (*bool or int*) −Whether to save intermediate compilation graphs. Default `0` . This parameter will be deprecated and removed in a future version. Please use the environment variable *MS_DEV_SAVE_GRAPHS* instead.
- **save_graphs_path** (*str*) −Path to save graphs. Default `"."`. This parameter will be deprecated and removed in a future version. Please use the environment variable *MS_DEV_SAVE_GRAPHS_PATH* instead.
- **precompile_only** (*bool*) −Whether to only precompile the network. Default `False` . This parameter will be deprecated and removed in a future version. Please use the environment variable *MS_DEV_PRECOMPILE_ONLY* instead.
- **enable_compile_cache** (*bool*) −Whether to save or load the compiled cache of the graph. Default `False` . This is an experimental prototype that is subject to change and/or deletion. This parameter will be deprecated and removed in a future version. Please use the environment variable *MS_COMPILER_CACHE_ENABLE* instead.
- **ascend_config** (*dict*) −Set the parameters specific to Ascend hardware platform.
  - precision_mode (str): Mixed precision mode setting. Default `"force_fp16"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_precision.precision_mode()* instead.
  - jit_compile (bool): Whether to select online compilation. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_tuning.op_compile()* instead.
  - matmul_allow_hf32 (bool): Whether to convert FP32 to HF32 for Matmul operators. Default `False`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_precision.matmul_allow_hf32()* instead.
  - conv_allow_hf32 (bool): Whether to convert FP32 to HF32 for Conv operators. Default `True`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_precision.conv_allow_hf32()* instead.
  - op_precision_mode (str): Path to config file of op precision mode. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_precision.op_precision_mode()* instead.
  - op_debug_option (str): Enable debugging options for Ascend operators. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.ascend.op_debug.debug_option()* instead.

  - ge_options (dict): Set options for CANN. This parameter will be deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
  - atomic_clean_policy (int): The policy for cleaning memory occupied by atomic operators in the network. Default `1` represents that memory is not cleaned centrally, `0` represents that memory is cleaned centrally. This parameter will be deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
  - exception_dump (str): Enable Ascend operator exception dump. Default `"2"` . This parameter has been deprecated and removed. Please use the api *mindspore.device_context.ascend.op_debug.aclinit_config()* instead.
  - host_scheduling_max_threshold(int): The max threshold to control whether the dynamic shape process is used when run the static graph. Default `0` . This parameter will be deprecated and removed in future versions. Please use the related parameter of *mindspore.jit()* instead.
  - parallel_speed_up_json_path(Union[str, None]): The path to the parallel speed up json file. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.parallel.auto_parallel.AutoParallel.transformer_opt()* instead.
  - hccl_watchdog (bool): Enable a thread to monitor the failure of collective communication. Default `True` .

- **gpu_config** (*dict*) –Set the parameters specific to gpu hardware platform. It is not set by default.
  - conv_fprop_algo (str): Specifies convolution forward algorithm. Default `"normal"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.gpu.op_tuning.conv_fprop_algo()* instead.
  - conv_dgrad_algo (str): Specifies convolution data grad algorithm. Default `"normal"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.gpu.op_tuning.conv_dgrad_algo()* instead.
  - conv_wgrad_algo (str): Specifies convolution filter grad algorithm. Default `"normal"` . This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.gpu.op_tuning.conv_wgrad_algo()* instead.
  - conv_allow_tf32 (bool): Controls to allow Tensor core TF32 computation on CUDNN. Default `True`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.gpu.op_precision.conv_allow_tf32()* instead.
  - matmul_allow_tf32 (bool): Controls to allow Tensor core TF32 computation on CUBLAS. Default `False`. This parameter will be deprecated and removed in future versions. Please use the api *mindspore.device_context.gpu.op_precision.matmul_allow_tf32()* instead.
- **print_file_path** (*str*) –This parameter will be deprecated and removed in future versions.
- **env_config_path** (*str*) –This parameter will be deprecated and removed in future versions.
- **debug_level** (*int*) –This parameter will be deprecated and removed in future versions.
- **reserve_class_name_in_scope** (*bool*) –This parameter will be deprecated and removed in future versions.
- **check_bprop** (*bool*) –This parameter will be deprecated and removed in future versions.
- **enable_reduce_precision** (*bool*) –This parameter will be deprecated and removed in a future versions.
- **grad_for_scalar** (*bool*) –This parameter will be deprecated and removed in future versions.
- **support_binary** (*bool*) –Whether to support run .pyc or .so in graph mode.

Examples

```
>>> import mindspore as ms
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> ms.set_context(precompile_only=True)
>>> ms.set_context(device_target="Ascend")
>>> ms.set_context(device_id=0)
>>> ms.set_context(save_graphs=True, save_graphs_path="./model.ms")
>>> ms.set_context(enable_reduce_precision=True)
>>> ms.set_context(reserve_class_name_in_scope=True)
>>> ms.set_context(variable_memory_max_size="6GB")
>>> ms.set_context(aoe_tune_mode="online")
>>> ms.set_context(aoe_config={"job_type": "2"})
>>> ms.set_context(check_bprop=True)
>>> ms.set_context(max_device_memory="3.5GB")
>>> ms.set_context(mempool_block_size="1GB")
>>> ms.set_context(print_file_path="print.pb")
>>> ms.set_context(max_call_depth=80)
>>> ms.set_context(env_config_path="./env_config.json")
>>> ms.set_context(grad_for_scalar=True)
>>> ms.set_context(enable_compile_cache=True, compile_cache_path="./cache.ms")
>>> ms.set_context(pynative_synchronize=True)
>>> ms.set_context(runtime_num_threads=10)
>>> ms.set_context(inter_op_parallel_num=4)
>>> ms.set_context(disable_format_transform=True)
>>> ms.set_context(memory_optimize_level='O0')
>>> ms.set_context(memory_offload='ON')
>>> ms.set_context(deterministic='ON')
>>> ms.set_context(ascend_config={"precision_mode": "force_fp16", "jit_compile": True,
...                 "atomic_clean_policy": 1, "op_precision_mode": "./op_precision_config_file
 ↪",
...                 "op_debug_option": "oom",
...                 "ge_options": {"global": {"ge.opSelectImplmode": "high_precision"},
...                                "session": {"ge.exec.atomicCleanPolicy": "0"}}})
>>> ms.set_context(jit_syntax_level=ms.STRICT)
>>> ms.set_context(debug_level=ms.context.DEBUG)
>>> ms.set_context(gpu_config={"conv_fprop_algo": "performance", "conv_allow_tf32": True,
...                 "matmul_allow_tf32": True})
>>> ms.set_context(jit_config={"jit_level": "O0"})
>>> ms.set_context(exec_order="bfs")
```

1.2.5 mindspore.get_context

mindspore.**get_context**(*attr_key*)

Get context attribute value according to the input key, this api will be deprecated and removed in future versions, please use *mindspore.get_current_device()* instead.

If some attributes are not set, they will be automatically obtained.

Parameters

attr_key (*str*) –The key of the attribute.

Returns

Object, The value of given attribute key.

Raises

ValueError –If input key is not an attribute in context.

Examples

```
>>> import mindspore as ms
>>> ms.get_context("device_target")
>>> ms.get_context("device_id")
```

1.2.6 mindspore.set_auto_parallel_context

mindspore.**set_auto_parallel_context**(***kwargs*)

Set auto parallel context, this api will be deprecated and removed in future versions, please use the api *mindspore.parallel.auto_parallel.AutoParallel* instead.

Note: CPU only support data parallel.

Some configurations are parallel mode specific, see the below table for details:

Common	AUTO_PARALLEL
device_num	gradient_fp32_sync
global_rank	loss_repeated_mean
gradients_mean	search_mode
parallel_mode	parameter_broadcast
all_reduce_fusion_config	strategy_ckpt_load_file
enable_parallel_optimizer	strategy_ckpt_save_file
parallel_optimizer_config	dataset_strategy
enable_alltoall	pipeline_stages
pipeline_config	auto_parallel_search_mode
force_fp32_communication	pipeline_result_broadcast
	comm_fusion
	strategy_ckpt_config
	group_ckpt_save_file
	auto_pipeline
	dump_local_norm
	dump_local_norm_path
	dump_device_local_norm

Parameters

- **device_num** (*int*) –Available device number, the value must be in [1, 4096]. Default: 1 .
- **global_rank** (*int*) –Global rank id, the value must be in [0, 4095]. Default: 0 .

- **gradients_mean** (*bool*) −Whether to perform mean operator after allreduce of gradients. "stand_alone" do not support gradients_mean. Default: `False` .
- **gradient_fp32_sync** (*bool*) −Run allreduce of gradients in fp32. "stand_alone", "data_parallel" and "hybrid_parallel" do not support gradient_fp32_sync. Default: `True` .
- **loss_repeated_mean** (*bool*) −calculation is repeated. Default: `True` .
- **parallel_mode** (*str*) −There are five kinds of parallel modes, `"stand_alone"` , `"data_parallel"` , `"hybrid_parallel"` , `"semi_auto_parallel"` and `"auto_parallel"` . Note the pynative mode only supports the `"stand_alone"` and `"data_parallel"` mode. Default: `"stand_alone"` .
  - stand_alone: Only one processor is working.
  - data_parallel: Distributes the data across different processors.
  - hybrid_parallel: Achieves data parallelism and model parallelism manually.
  - semi_auto_parallel: Achieves data and model parallelism by setting parallel strategies.
  - auto_parallel: Achieving parallelism automatically.
- **search_mode** (*str*) −There are three kinds of shard strategy search modes: `"recursive_programming"` , `"sharding_propagation"` and `"dynamic_programming"` (Not recommended). Only works in `"auto_parallel"` mode. Default: `"recursive_programming"` .
  - recursive_programming: Recursive programming search mode. In order to obtain optimal performance, it is recommended that users set the batch size to be greater than or equal to the product of the number of devices and the number of multi-copy parallelism.
  - sharding_propagation: Propagate shardings from configured ops to non-configured ops. Dynamic shapes are not supported currently.
  - dynamic_programming: Dynamic programming search mode.
- **auto_parallel_search_mode** (*str*) −This is the old version of 'search_mode'. Here, remaining this attribute is for forward compatibility, and this attribute will be deleted in a future MindSpore version.
- **parameter_broadcast** (*bool*) −Whether to broadcast parameters before training. Before training, in order to have the same network initialization parameter values for all devices, broadcast the parameters on device 0 to other devices. Parameter broadcasting in different parallel modes is different, `data_parallel` mode, all parameters are broadcast except for the parameter whose attribute layerwise_parallel is `True` . `Hybrid_parallel` , `semi_auto_parallel` and `auto_parallel mode` , the segmented parameters do not participate in broadcasting. Default: `False` .
- **strategy_ckpt_load_file** (*str*) −The path to load parallel strategy checkpoint. The parameter is not to be recommended currently, it is better using 'strategy_ckpt_config' to replace it. Default: `''`
- **strategy_ckpt_save_file** (*str*) −The path to save parallel strategy checkpoint. The parameter is not to be recommended currently, it is better using 'strategy_ckpt_config' to replace it. Default: `''`
- **full_batch** (*bool*) −If you load whole batch datasets in `auto_parallel` mode, this parameter should be set as `True` . Default: `False` . The interface is not to be recommended currently, it is better using 'dataset_strategy' to replace it.
- **dataset_strategy** (*Union[str, tuple]*) −Dataset sharding strategy. Default: `"data_parallel"` . dataset_strategy="data_parallel" is equal to full_batch=False, dataset_strategy="full_batch" is equal to full_batch=True. For execution mode is 'GRAPH_MODE' and dataset load into net by model parallel strategy likes ds_stra ((1, 8), (1, 8)), it requires using set_auto_parallel_context(dataset_strategy=ds_stra). The dataset sharding strategy is not affected by the currently configured parallel mode. parallel strategy also supports tuple of Layout.
- **enable_parallel_optimizer** (*bool*) −This is a developing feature, which shards the weight update computation for data parallel training in the benefit of time and memory saving. Currently, auto and semi auto parallel mode

support all optimizers in both Ascend and GPU. Data parallel mode only supports *Lamb* and *AdamWeightDecay* in Ascend . Default: `False` .

- **force_fp32_communication** (*bool*) –A switch that determines whether reduce operators (AllReduce, ReduceScatter) are forced to use the fp32 data type for communication during communication. True is the enable switch. Default: `False` .
- **enable_alltoall** (*bool*) –A switch that allows AllToAll operators to be generated during communication. If its value is `False` , there will be a combination of operators such as AllGather, Split and Concat instead of AllToAll. Default: `False` .
- **all_reduce_fusion_config** (*list*) –Set allreduce fusion strategy by parameters indices. Only support ReduceOp.SUM and HCCL_WORLD_GROUP/NCCL_WORLD_GROUP. No Default, if it is not set, the fusion is closed.
- **pipeline_stages** (*int*) –Set the stage information for pipeline parallel. This indicates how the devices are distributed alone in the pipeline. The total devices will be divided into 'pipeline_stags' stages. Default: `1` .
- **pipeline_result_broadcast** (*bool*) –A switch that broadcast the last stage result to all other stage in pipeline parallel inference. Default: `False` .
- **pipeline_config** (*dict*) –A dict contains the keys and values for setting the pipeline parallelism configuration. It supports the following keys:
  - pipeline_interleave(bool): Indicates whether to enable the interleaved execution mode.
  - pipeline_scheduler(str): Indicates the scheduling mode for pipeline parallelism. Only support `gpipe/1f1b/seqpipe/seqvpp/seqsmartvpp`. When applying seqsmartvpp, the pipeline parallel must be an even number.
- **parallel_optimizer_config** (*dict*) –A dict contains the keys and values for setting the parallel optimizer configure. The configure provides more detailed behavior control about parallel training when parallel optimizer is enabled. The configure will be effective when we use mindspore.set_auto_parallel_context(enable_parallel_optimizer=True). It supports the following keys.
  - gradient_accumulation_shard(bool): Please using optimizer_level: `level2` to replace this config. If `true` , the accumulation gradient parameters will be sharded across the data parallel devices. This will introduce additional communication(ReduceScatter) at each step when accumulate the gradients, but saves a lot of device memories, thus can make model be trained with larger batch size. This configure is effective only when the model runs on pipeline training or gradient accumulation with data parallel. Default `False` .
  - parallel_optimizer_threshold(int): Set the threshold of parallel optimizer. When parallel optimizer is enabled, parameters with size smaller than this threshold will not be sharded across the devices. Parameter size is calculated as: shape[0] * ···* shape[n] * size(dtype). Non-negative. Unit: KB. Default: `64` .
  - optimizer_weight_shard_size(int): Set the optimizer weight shard group size, if you want to specific the maximum group size across devices when the parallel optimizer is enabled. The numerical range can be (0, device_num]. If pipeline parallel is enabled, the numerical range is (0, device_num/stage]. If the size of data parallel communication domain of the parameter cannot be divided by *optimizer_weight_shard_size*, then the specified communication group size will not take effect. Default value is `-1` , which means the optimizer weight shard group size will be the size of data parallel group of each parameter.
  - optimizer_level(str, optional): optimizer_level configuration is used to specify the splitting level for optimizer sharding. It is important to note that the implementation of optimizer sharding in static graph is inconsistent with dynamic graph like megatron, but the memory optimization effect is the same. When optimizer_level= `level1` , splitting is performed on weights and optimizer state. When optimizer_level= `level2` , splitting is performed on weights, optimizer state, and gradients. When optimizer_level= `level3` , splitting is performed on weights, optimizer state, gradients, additionally, before the backward pass, the weights are further applied with allgather communication to release the memory used by the forward pass allgather. It must be one of [`level1`, `level2`, `level3`]. Default: `level1`.

- **comm_fusion** (*dict*) −A dict contains the types and configurations for setting the communication fusion. each communication fusion config has two keys: "mode" and "config". It supports following communication fusion types and configurations:
  - openstate: Whether turn on the communication fusion or not. If *openstate* is `True` , turn on the communication fusion, otherwise, turn off the communication fusion. Default: `True` .
  - allreduce: If communication fusion type is *allreduce*. The *mode* contains: *auto*, *size* and *index*. In *auto* mode, AllReduce fusion is configured by gradients size and the default fusion threshold is *64* MB. In 'size' mode, AllReduce fusion is configured by gradients size manually, and the fusion threshold must be larger than *0* MB. In *index* mode, it is same as *all_reduce_fusion_config*.
  - allgather: If communication fusion type is *allgather*. The *mode* contains: *auto*, *size*. In *auto* mode, AllGather fusion is configured by gradients size, and the default fusion threshold is *64* MB. In 'size' mode, AllGather fusion is configured by gradients size manually, and the fusion threshold must be larger than *0* MB.
  - reducescatter: If communication fusion type is *reducescatter*. The *mode* contains: *auto* and *size*. Config is same as *allgather*.
- **strategy_ckpt_config** (*dict*) −A dict contains the configurations for setting the parallel strategy file. This interface contains the functions of parameter *strategy_ckpt_load_file* and *strategy_ckpt_save_file*, it is recommonded to use this parameter to replace those two parameters. It contains following configurations:
  - load_file (str): The path to load parallel strategy checkpoint. If the file name extension is *.json*, the file is loaded in JSON format. Otherwise, the file is loaded in ProtoBuf format. Default: `''`
  - save_file (str): The path to save parallel strategy checkpoint. If the file name extension is *.json*, the file is saved in JSON format. Otherwise, the file is saved in ProtoBuf format. Default: `''`
  - only_trainable_params (bool): Only save/load the strategy information for trainable parameter. Default: `True` .
- **group_ckpt_save_file** (*str*) −The path to save parallel group checkpoint.
- **auto_pipeline** (*bool*) −Set the pipeline stage number to automatic. Its value will be selected between 1 and the parameter *pipeline_stages*. This option requires the *parallel_mode* to be `auto_parallel` and the *search_mode* to be `recursive_programming`. Default: `False` .
- **dump_local_norm** (*bool*) −Whether to dump local_norm value, when the *parallel_mode* is set to `semi_auto_parallel` or `auto_parallel`. Default: `False` .
- **dump_local_norm_path** (*str*) −The path to save dump files of local_norm value. Default: `''` .
- **dump_device_local_norm** (*bool*) −Whether to dump device_local_norm value, when the *parallel_mode* is set to `semi_auto_parallel` or `auto_parallel`. Default: `False` .

Raises

ValueError −If input key is not attribute in auto parallel context.

Examples

```
>>> import mindspore as ms
>>> ms.set_auto_parallel_context(device_num=8)
>>> ms.set_auto_parallel_context(global_rank=0)
>>> ms.set_auto_parallel_context(gradients_mean=True)
>>> ms.set_auto_parallel_context(gradient_fp32_sync=False)
>>> ms.set_auto_parallel_context(parallel_mode="auto_parallel")
>>> ms.set_auto_parallel_context(search_mode="recursive_programming")
>>> ms.set_auto_parallel_context(auto_parallel_search_mode="recursive_programming")
>>> ms.set_auto_parallel_context(parameter_broadcast=False)
```

(continues on next page)

(continued from previous page)

```
>>> ms.set_auto_parallel_context(strategy_ckpt_load_file="./strategy_stage1.ckpt")
>>> ms.set_auto_parallel_context(strategy_ckpt_save_file="./strategy_stage1.ckpt")
>>> ms.set_auto_parallel_context(dataset_strategy=((1, 8), (1, 8)))
>>> ms.set_auto_parallel_context(enable_parallel_optimizer=False)
>>> ms.set_auto_parallel_context(enable_alltoall=False)
>>> ms.set_auto_parallel_context(all_reduce_fusion_config=[8, 160])
>>> ms.set_auto_parallel_context(pipeline_stages=2)
>>> ms.set_auto_parallel_context(pipeline_stages=2, pipeline_result_broadcast=True)
>>> parallel_config = {"gradient_accumulation_shard": True, "parallel_optimizer_threshold":␣
↪24,
...                     "optimizer_weight_shard_size": 2, "optimizer_level": "level3"}
>>> ms.set_auto_parallel_context(parallel_optimizer_config=parallel_config, enable_parallel_
↪optimizer=True)
>>> config = {"allreduce": {"mode": "size", "config": 32}, "allgather": {"mode": "size",
↪"config": 32}}
>>> ms.set_auto_parallel_context(comm_fusion=config)
>>> stra_ckpt_dict = {"load_file": "./stra0.ckpt", "save_file": "./stra1.ckpt", "only_
↪trainable_params": False}
>>> ms.set_auto_parallel_context(strategy_ckpt_config=stra_ckpt_dict)
```

1.2.7 mindspore.get_auto_parallel_context

mindspore.**get_auto_parallel_context**(*attr_key*)

Get auto parallel context attribute value according to the key, this api will be deprecated and removed in future versions.

Parameters

attr_key (*str*) –The key of the attribute.

Returns

Returns attribute value according to the key.

Raises

ValueError –If input key is not attribute in auto parallel context.

Examples

```
>>> import mindspore as ms
>>> parallel_mode = ms.get_auto_parallel_context("parallel_mode")
>>> dataset_strategy = ms.get_auto_parallel_context("dataset_strategy")
```

1.2.8 mindspore.reset_auto_parallel_context

mindspore.**reset_auto_parallel_context**()

Reset auto parallel context attributes to the default values, this api will be deprecated and removed in future versions, please use the api *mindspore.parallel.auto_parallel.AutoParallel* instead.

- device_num: 1.
- global_rank: 0.
- gradients_mean: False.

- gradient_fp32_sync: True.
- parallel_mode: 'stand_alone'.
- search_mode: 'recursive_programming'.
- auto_parallel_search_mode: 'recursive_programming'.
- parameter_broadcast: False.
- strategy_ckpt_load_file: ''.
- strategy_ckpt_save_file: ''.
- full_batch: False.
- enable_parallel_optimizer: False.
- force_fp32_communication: False.
- enable_alltoall: False.
- pipeline_stages: 1.
- pipeline_result_broadcast: False.
- fusion_threshold: 64.
- auto_pipeline: False.
- dump_local_norm: False.
- dump_local_norm_path: ''.
- dump_device_local_norm: False.

Examples

```
>>> import mindspore as ms
>>> ms.reset_auto_parallel_context()
```

1.2.9 mindspore.ParallelMode

class mindspore.**ParallelMode**

Parallel mode options.

There are five kinds of parallel modes, `STAND_ALONE`, `DATA_PARALLEL`, `HYBRID_PARALLEL`, `SEMI_AUTO_PARALLEL` and `AUTO_PARALLEL`. Default: `STAND_ALONE`.

- `STAND_ALONE`: Only one processor is working.
- `DATA_PARALLEL`: Distributes the data across different processors.
- `HYBRID_PARALLEL`: Achieves data parallelism and model parallelism manually.
- `SEMI_AUTO_PARALLEL`: Achieves data parallelism and model parallelism by setting parallel strategies.
- `AUTO_PARALLEL`: Achieves parallelism automatically.

`MODE_LIST`: The list of all supported parallel modes.

1.2.10 mindspore.set_ps_context

mindspore.**set_ps_context**(***kwargs*)

Set parameter server training mode context, this api will be deprecated and removed in future versions.

Note: Parameter server mode is only supported in graph mode. Some other environment variables should also be set for parameter server training mode. These environment variables are listed below:

- MS_SERVER_NUM: Server number
- MS_WORKER_NUM: Worker number
- MS_SCHED_HOST: Scheduler IP address
- MS_SCHED_PORT: Scheduler port
- MS_ROLE: The role of this process:
  - MS_SCHED: represents the scheduler,
  - MS_WORKER: represents the worker,
  - MS_PSERVER/MS_SERVER: represents the Server

Parameters

- **enable_ps** (*bool*) –Whether to enable parameter server training mode. Only after enable_ps is set True, the environment variables will be effective. Default: `False` .
- **config_file_path** (*str*) –Configuration file path used by recovery, parameter server training mode only supports Server disaster recovery currently. Default: `''` .
- **scheduler_manage_port** (*int*) –Scheduler manage port used to scale out/in. Default: `11202` .
- **enable_ssl** (*bool*) –Set PS SSL mode enabled or disabled. Default: `False` .
- **client_password** (*str*) –Password to decrypt the secret key stored in the client certificate. Default: `''` .
- **server_password** (*str*) –Password to decrypt the secret key stored in the server certificate. Default: `''` .

Raises

ValueError –If input key is not the attribute in parameter server training mode context.

Examples

```
>>> import mindspore as ms
>>> ms.set_ps_context(enable_ps=True, enable_ssl=True, client_password='123456', server_
↪password='123456')
```

1.2.11 mindspore.get_ps_context

mindspore.**get_ps_context**(*attr_key*)

Get parameter server training mode context attribute value according to the key, this api will be deprecated and removed in future versions.

Parameters

attr_key (*str*) –The key of the attribute:

- enable_ps (bool, optional): Whether to enable parameter server training mode. Default: `False` .
- config_file_path (str, optional): Configuration file path used by recovery, parameter server training mode only supports Server disaster recovery currently. Default: `''` .
- scheduler_manage_port (int, optional): Scheduler manage port used to scale out/in. Default: `11202` .
- enable_ssl (bool, optional): Set PS SSL mode enabled or disabled. Default: `False` .
- client_password (str, optional): Password to decrypt the secret key stored in the client certificate. Default: `''` .
- server_password (str, optional): Password to decrypt the secret key stored in the server certificate. Default: `''` .

Returns

Returns attribute value according to the key.

Raises

ValueError –If input key is not attribute in auto parallel context.

Examples

```
>>> import mindspore as ms
>>> ms.get_ps_context("enable_ps")
```

1.2.12 mindspore.reset_ps_context

mindspore.**reset_ps_context**()

Reset parameter server training mode context attributes to the default values, this api will be deprecated and removed in future versions.

Meaning of each field and its default value refer to *mindspore.set_ps_context()*.

Examples

```
>>> import mindspore as ms
>>> ms.reset_ps_context()
```

1.2.13 mindspore.set_algo_parameters

mindspore.**set_algo_parameters**(***kwargs*)

Set parameters in the algorithm for parallel strategy searching.

Note: The attribute name is required. This interface works ONLY in AUTO_PARALLEL mode.

Parameters

- **fully_use_devices** (*bool*) −Whether ONLY searching strategies that fully use all available devices. Default: `False` . For example with 8 devices available, if set `True` , strategy (4, 1) will not be included in ReLU's candidate strategies, because strategy (4, 1) only utilizes 4 devices.
- **elementwise_op_strategy_follow** (*bool*) −Whether the elementwise operator has the consistent strategies as its subsequent operators. Elementwise operators refer to operators that operate on input element by element, such as Add, ReLU, etc. Default: `False` . For the example of ReLU followed by Add, if this flag is set `True` , then the searched strategy by the algorithm guarantees that strategies of these two operators are consistent, e.g., ReLU's strategy (8, 1) and Add's strategy ((8, 1), (8, 1)).
- **enable_algo_approxi** (*bool*) −Whether to enable the approximation in the algorithms. Default: `False` . Due to large solution space in searching parallel strategy for large DNN model, the algorithm takes fairly long time in this case. To mitigate it, if this flag is set `True` , an approximation is made to discard some candidate strategies, so that the solution space is shrunken.
- **algo_approxi_epsilon** (*float*) −The epsilon value used in the approximation algorithm. Default: `0.1` . This value describes the extent of approximation. For example, the number of candidate strategies of an operator is S, if 'enable_algo_approxi' is `True` , then the remaining strategies is of size: min{S, 1/epsilon}.
- **tensor_slice_align_enable** (*bool*) −Whether to check the shape of tensor slice of MatMul. Default: `False` . Due to properties of some hardware, MatMul kernel only with large shapes can show advantages. If this flag is `True` , then the slice shape of MatMul is checked to prevent irregular shapes.
- **tensor_slice_align_size** (*int*) −The minimum tensor slice shape of MatMul, the value must be in [1, 1024]. Default: `16` . If 'tensor_slice_align_enable' is set `True` , then the slice size of last dimension of MatMul tensors should be multiple of this value.

Raises

ValueError −If context keyword is not recognized.

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Please see the rank table startup for more details.

For the GPU devices, users need to prepare the host file and mpi, please see the mpirun startup .

For the CPU device, users need to write a dynamic cluster startup script, please see the Dynamic Cluster Startup .

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import nn, ops, train
```

(continues on next page)

(continued from previous page)

```
>>> from mindspore.communication import init
>>> from mindspore.common.initializer import initializer
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.AUTO_PARALLEL,
...                              search_mode="sharding_propagation")
>>> init()
>>> ms.set_algo_parameters(fully_use_devices=True)
>>> ms.set_algo_parameters(elementwise_op_strategy_follow=True)
>>> ms.set_algo_parameters(enable_algo_approxi=True)
>>> ms.set_algo_parameters(algo_approxi_epsilon=0.2)
>>> ms.set_algo_parameters(tensor_slice_align_enable=True)
>>> ms.set_algo_parameters(tensor_slice_align_size=8)
>>>
>>> # Define the network structure.
>>> class Dense(nn.Cell):
...     def __init__(self, in_channels, out_channels):
...         super().__init__()
...         self.weight = ms.Parameter(initializer("normal", [in_channels, out_channels], ms.
↪float32))
...         self.bias = ms.Parameter(initializer("normal", [out_channels], ms.float32))
...         self.matmul = ops.MatMul()
...         self.add = ops.Add()
...
...     def construct(self, x):
...         x = self.matmul(x, self.weight)
...         x = self.add(x, self.bias)
...         return x
>>>
>>> class FFN(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.flatten = ops.Flatten()
...         self.dense1 = Dense(28*28, 64)
...         self.relu = ops.ReLU()
...         self.dense2 = Dense(64, 10)
...
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.dense1(x)
...         x = self.relu(x)
...         x = self.dense2(x)
...         return x
>>> net = FFN()
>>> net.dense1.matmul.shard(((2, 1), (1, 2)))
>>>
>>> # Create dataset.
>>> step_per_epoch = 16
>>> def get_dataset(*inputs):
...     def generate():
...         for _ in range(step_per_epoch):
...             yield inputs
...     return generate
>>>
>>> input_data = np.random.rand(1, 28, 28).astype(np.float32)
>>> label_data = np.random.rand(1).astype(np.int32)
>>> fake_dataset = get_dataset(input_data, label_data)
```

(continues on next page)

(continued from previous page)

```
>>> dataset = ds.GeneratorDataset(fake_dataset, ["input", "label"])
>>> # Train network.
>>> optimizer = nn.Momentum(net.trainable_params(), 1e-3, 0.1)
>>> loss_fn = nn.CrossEntropyLoss()
>>> loss_cb = train.LossMonitor()
>>> model = ms.Model(network=net, loss_fn=loss_fn, optimizer=optimizer)
>>> model.train(epoch=2, train_dataset=dataset, callbacks=[loss_cb])
```

1.2.14 mindspore.get_algo_parameters

mindspore.**get_algo_parameters**(*attr_key*)

Get the algorithm parameter config attributes.

Note: The attribute name is required. This interface works ONLY in AUTO_PARALLEL mode.

Parameters

attr_key (*str*) –The key of the attribute. The keys include: "fully_use_devices", "elementwise_op_strategy_follow", "enable_algo_approxi", "algo_approxi_epsilon", "tensor_slice_align_enable","tensor_slice_align_size". See *mindspore.set_algo_parameters()* for more details about the meaning of the attributes.

Returns

Return attribute value according to the key.

Raises

ValueError –If context keyword is not recognized.

Examples

```
>>> import mindspore as ms
>>> ms.get_algo_parameters("fully_use_devices")
False
```

1.2.15 mindspore.reset_algo_parameters

mindspore.**reset_algo_parameters**()

Reset the algorithm parameter attributes.

Note: This interface works ONLY in AUTO_PARALLEL mode.

After reset, the values of the attributes are:

- fully_use_devices: False.
- elementwise_op_strategy_follow: False.
- enable_algo_approxi: False.
- algo_approxi_epsilon: 0.1.

- tensor_slice_align_enable: False.
- tensor_slice_align_size: 16.

Examples

```
>>> import mindspore as ms
>>> ms.reset_algo_parameters()
```

1.2.16 mindspore.set_offload_context

mindspore.**set_offload_context**(*offload_config*)

Configure heterogeneous training detailed parameters to adjust the offload strategy, this api will be deprecated and removed in future versions.

Note: The offload configuration is only used if the memory offload feature is enabled via mindspore.set_context(memory_offload="ON"), and the memory_optimize_level must be set to O0. On the Ascend hardware platform, the graph compilation level must be O0.

Parameters

offload_config (*dict*) – A dict contains the keys and values for setting the offload context configure.It supports the following keys.

- offload_path (str): The path of offload, relative paths are supported. Default: "./offload".
- offload_cpu_size (str): The cpu memory size for offload. The format is "xxGB".
- offload_disk_size (str): The disk size for offload. The format is "xxGB"
- hbm_ratio (float): The ratio that can be used based on the maximum device memory. The range is (0,1], Default: 1.0.
- cpu_ratio (float): The ratio that can be used based on the maximum host memory. The range is (0,1], Default: 1.0.
- enable_pinned_mem (bool): The flag of whether enabling Pinned Memory. Default: True.
- enable_aio (bool): The flag of whether enabling aio. Default: True.
- aio_block_size (str): The size of aio block. The format is "xxGB".
- aio_queue_depth (int): The depth of aio queue.
- offload_param (str): The param for offload destination, cpu or disk, Default: "".
- offload_checkpoint (str): The checkpoint for offload destination, only valid if recompute is turned on, cpu or disk, Default: "".
- auto_offload (bool): The flag of whether auto offload. Default: True.
- host_mem_block_size (str): The memory block size of host memory pool. The format is "xxGB"

Raises

ValueError – If input key is not attribute in auto parallel context.

Examples

```
>>> from mindspore import context
>>> context.set_offload_context(offload_config={"offload_param":"cpu"})
```

1.2.17 mindspore.get_offload_context

mindspore.**get_offload_context**()

Gets the offload configuration parameters, this api will be deprecated and removed in future versions.

Configure through interface mindspore.set_offload_context(). If the user is not set, the default configuration is obtained.

Returns

Dict, heterogeneous training offload detailed configuration parameters.

Examples

```
>>> from mindspore import context
>>> offload_config = context.get_offload_context()
```

1.3 Seed

mindspore.set_seed	Set global seed.
mindspore.get_seed	Get global seed.

1.3.1 mindspore.set_seed

mindspore.**set_seed**(*seed*)

Set global seed.

Note:

- The global seed is used by numpy.random, mindspore.common.Initializer and mindspore.nn.probability.distribution.
- If global seed is not set, these packages will use their own default seed independently, numpy.random and mindspore.common.Initializer will choose a random seed, mindspore.nn.probability.distribution will use zero.
- Seed set by numpy.random.seed() only used by numpy.random, while seed set by this API will also used by numpy.random, so just set all seed by this API is recommended.
- In semi_auto_parallel/auto_parallel mode, when using set_seed, weights with same shape and same sharding strategy in the same device would be initialized to the same result, otherwise, they would be initialized to the different result.

Parameters

seed (*int*) – The seed to be set.

Raises

- **ValueError** – If seed is invalid (< 0).
- **TypeError** – If seed isn't an int.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, set_seed, Parameter, ops
>>> from mindspore.common.initializer import initializer
>>> # Note: (1) Please make sure the code is running in PYNATIVE MODE;
>>> # (2) Because Composite-level ops need parameters to be Tensors, for below examples,
>>> # when using ops.uniform operator, minval and maxval are initialised as:
>>> minval = Tensor(1.0, ms.float32)
>>> maxval = Tensor(2.0, ms.float32)
>>>
>>> # 1. If global seed is not set, numpy.random and initializer will choose a random seed:
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A1
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A2
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W1
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W2
>>> # Rerun the program will get different results:
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A3
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A4
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W3
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W4
>>>
>>> # 2. If global seed is set, numpy.random and initializer will use it:
>>> set_seed(1234)
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A1
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A2
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W1
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W2
>>> # Rerun the program will get the same results:
>>> set_seed(1234)
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A1
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A2
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W1
>>> w1 = Parameter(initializer("uniform", [2, 2], ms.float32), name="w1") # W2
>>>
>>> # 3. If neither global seed nor op seed is set, mindspore.ops.function.random_func and
>>> # mindspore.nn.probability.distribution will choose a random seed:
>>> c1 = ops.uniform((1, 4), minval, maxval) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval) # C2
>>> # Rerun the program will get different results:
>>> c1 = ops.uniform((1, 4), minval, maxval) # C3
>>> c2 = ops.uniform((1, 4), minval, maxval) # C4
>>>
>>> # 4. If global seed is set, but op seed is not set, mindspore.ops.function.random_func␣
 ↪and
>>> # mindspore.nn.probability.distribution will calculate a seed according to global seed␣
 ↪and
>>> # default op seed. Each call will change the default op seed, thus each call get␣
 ↪different
>>> # results.
>>> set_seed(1234)
>>> c1 = ops.uniform((1, 4), minval, maxval) # C1
```

(continues on next page)

(continued from previous page)

```
>>> c2 = ops.uniform((1, 4), minval, maxval) # C2
>>> # Rerun the program will get the same results:
>>> set_seed(1234)
>>> c1 = ops.uniform((1, 4), minval, maxval) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval) # C2
>>>
>>> # 5. If both global seed and op seed are set, mindspore.ops.function.random_func and
>>> # mindspore.nn.probability.distribution will calculate a seed according to global seed␣
 ↪and
>>> # op seed counter. Each call will change the op seed counter, thus each call get␣
 ↪different
>>> # results.
>>> set_seed(1234)
>>> c1 = ops.uniform((1, 4), minval, maxval, seed=2) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval, seed=2) # C2
>>> # Rerun the program will get the same results:
>>> set_seed(1234)
>>> c1 = ops.uniform((1, 4), minval, maxval, seed=2) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval, seed=2) # C2
>>>
>>> # 6. If op seed is set but global seed is not set, 0 will be used as global seed. Then
>>> # mindspore.ops.function.random_func and mindspore.nn.probability.distribution act as in
>>> # condition 5.
>>> c1 = ops.uniform((1, 4), minval, maxval, seed=2) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval, seed=2) # C2
>>> # Rerun the program will get the different results:
>>> c1 = ops.uniform((1, 4), minval, maxval, seed=2) # C1
>>> c2 = ops.uniform((1, 4), minval, maxval, seed=2) # C2
>>>
>>> # 7. Recall set_seed() in the program will reset numpy seed and op seed counter of
>>> # mindspore.ops.function.random_func and mindspore.nn.probability.distribution.
>>> set_seed(1234)
>>> np_1 = np.random.normal(0, 1, [1]).astype(np.float32) # A1
>>> c1 = ops.uniform((1, 4), minval, maxval, seed=2) # C1
>>> set_seed(1234)
>>> np_2 = np.random.normal(0, 1, [1]).astype(np.float32) # still get A1
>>> c2 = ops.uniform((1, 4), minval, maxval, seed=2) # still get C1
```

1.3.2 mindspore.get_seed

mindspore.**get_seed**()

Get global seed.

Returns

Integer. The global seed.

Examples

```
>>> import mindspore as ms
>>> ms.set_seed(1234)
>>> seed = ms.get_seed()
>>> print(seed)
1234
```

1.4 Random Number Generator

mindspore.get_rng_state	Get the state of the default generator.
mindspore.Generator	A generator that manages the state of random numbers and provides seed and offset for random functions.
mindspore.initial_seed	Return the initial seed of the default generator.
mindspore.manual_seed	Set the default generator seed.
mindspore.seed	Seed the default generator with random number.
mindspore.set_rng_state	Set the state of the default generator.

1.4.1 mindspore.get_rng_state

mindspore.**get_rng_state**()

Get the state of the default generator.

Returns

Tensor, generator state.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import get_rng_state
>>> state = get_rng_state()
```

1.4.2 mindspore.Generator

class mindspore.**Generator**

A generator that manages the state of random numbers and provides seed and offset for random functions. When the seed and offset are fixed, the random function generates the same random sequence.

Note: Graph mode does not support the use of multiple generators at the same time for now.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import Generator
>>> generator = Generator()
>>> generator.manual_seed(5)
>>> print(generator.initial_seed())
5
>>> state = generator.get_state()
>>> generator.seed()
>>> generator.set_state(state)
>>> print(generator.initial_seed())
5
```

get_state()
Get the generator state.

Returns

Tensor, generator state.

initial_seed()
Return the initial seed of generator.

Returns

The initial seed of generator.

manual_seed(*seed*)
Set the generator seed.

Parameters

seed (*int*) – Set the generator seed.

Returns

Generator, the generator instance.

seed()
Seed generator with random number.

Returns

Randomly generated seeds, the type is int.

set_state(*state*)
Sets the generator state.

Parameters

state (*tensor*) – target state of the generator.

1.4.3 mindspore.initial_seed

mindspore.**initial_seed**()

Return the initial seed of the default generator.

Returns

The initial seed of the default generator.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import manual_seed, initial_seed
>>> manual_seed(14)
>>> print(initial_seed())
14
```

1.4.4 mindspore.manual_seed

mindspore.**manual_seed**(*seed*)

Set the default generator seed.

Parameters

seed (*int*) – Set the default generator seed.

Returns

Generator, the default generator.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import manual_seed, initial_seed
>>> manual_seed(13)
>>> print(initial_seed())
13
```

1.4.5 mindspore.seed

mindspore.**seed**()

Seed the default generator with random number.

Returns

Randomly generated seeds, the type is int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import seed
>>> print(seed())
1663920602
```

1.4.6 mindspore.set_rng_state

mindspore.**set_rng_state**(*state*)

Set the state of the default generator.

Parameters

state (Tensor) – the target state

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import set_rng_state, get_rng_state
>>> state = get_rng_state()
>>> set_rng_state(state)
```

1.5 Serialization

mindspore.async _ckpt_thread_status	Get the status of asynchronous save checkpoint thread.
mindspore.check_checkpoint	Check whether the checkpoint is valid.
mindspore.ckpt_to_safetensors	Converts MindSpore checkpoint files into safetensors format and saves them to *save_path*.
mindspore.convert_model	Convert mindir model to other format model.
mindspore.export	Export the MindSpore network into an offline model in the specified format.
mindspore.get _ckpt_path_with_strategy	Find available checkpoint file path from all backup checkpoint files of current rank.

continues on next page

Table 8 – continued from previous page

mindspore.load	Load MindIR.
mindspore.load_checkpoint	Load checkpoint info from a specified file.
mindspore.load_checkpoint_async	Load checkpoint info from a specified file asyncly.
mindspore.load_mindir	load protobuf file.
mindspore.load_param_into_net	Load parameters into network, return parameter list that are not loaded in the network.
mindspore.parse_print	Parse data file generated by *mindspore.ops.Print*.
mindspore.safetensors_to_ckpt	Converts safetensors files into MindSpore checkpoint format and saves them to *save_path*.
mindspore.save_checkpoint	Save checkpoint to a specified file.
mindspore.save_mindir	save protobuf file.

1.5.1 mindspore.async_ckpt_thread_status

mindspore.**async_ckpt_thread_status**()

Get the status of asynchronous save checkpoint thread.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

When performing asynchronous save checkpoint, you can determine whether the asynchronous thread is completed.

Returns

bool, True, Asynchronous save checkpoint thread is running. False, Asynchronous save checkpoint thread is not executing.

Examples

```
>>> import mindspore as ms
>>> ms.async_ckpt_thread_status()
False
```

1.5.2 mindspore.check_checkpoint

mindspore.**check_checkpoint**(*ckpt_file_name*)

Check whether the checkpoint is valid.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

Parameters

ckpt_file_name (*str*) – Checkpoint file name.

Returns

bool, whether the checkpoint is valid.

Examples

```
>>> import mindspore as ms
>>> ckpt_file_name = "./checkpoint/LeNet5-1_32.ckpt"
>>> check_result = ms.check_checkpoint(ckpt_file_name)
>>> print(check_result)
True
```

1.5.3 mindspore.ckpt_to_safetensors

mindspore.**ckpt_to_safetensors**(*file_path*, *save_path=None*, *name_map=None*, *file_name_regex=None*, *processes_num=1*)

Converts MindSpore checkpoint files into safetensors format and saves them to *save_path*. Safetensors is a reliable and portable machine learning model storage format introduced by Huggingface, used for securely storing Tensors with fast speed (zero copy).

Note: The number of multiprocess settings is related to the size of the host, and it is not recommended to set it too large, otherwise it may cause freezing. The safetensors format does not support the enc verification function. If ckpt is enabled to save enc verification, an error will be generated when performing the conversion. The safetensors format currently does not support crc verification function. If ckpt contains crc verification information, the crc verification information will be lost after conversion to safetensors.

Parameters

- **file_path** (*str*) –Path to the directory containing checkpoint files or a single checkpoint file (.ckpt).
- **save_path** (*str, optional*) –Directory path where safetensors files will be saved. Defaults: `None`.
- **name_map** (*dict, optional*) –Dictionary mapping original parameter names to new names. Defaults: `None`.
- **file_name_regex** (*str, optional*) –Regular expression used to match the file that needs to be converted. Defaults: `None`.
- **processes_num** (*int, optional*) –Number of processes to use for parallel processing. Defaults: 1.

Raises

ValueError –If the input path is invalid or the save_path is not a directory, or the file_path does not end with '.ckpt'.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> ms.ckpt_to_safetensors("./ckpt_save_path")
>>> ms.ckpt_to_safetensors("./ckpt_save_path/rank0/checkpoint_0.ckpt")
>>> ms.ckpt_to_safetensors(file_path="./ckpt_save_path/rank0/checkpoint_0.ckpt", save_path="./new_path/")
>>> namemap = {"lin.weight":"new_name"}
>>> ms.ckpt_to_safetensors("./ckpt_save_path/rank0/checkpoint_0.ckpt", "./new_path/", namemap)
```

1.5.4 mindspore.convert_model

mindspore.**convert_model**(*mindir_file*, *convert_file*, *file_format*)

Convert mindir model to other format model. The current version only supports conversion to ONNX models.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

Parameters

- **mindir_file** (*str*) –MindIR file name.
- **convert_file** (*str*) –Convert model file name.
- **file_format** (*str*) –Convert model's format, current version only supports "ONNX".

Raises

- **TypeError** –If the parameter *mindir_file* is not *str*.
- **TypeError** –If the parameter *convert_file* is not *str*.
- **ValueError** –If the parameter *file_format* is not "ONNX".

Examples

```
>>> import mindspore as ms
>>> ms.convert_model("lenet.mindir", "lenet.onnx", "ONNX")
```

1.5.5 mindspore.export

mindspore.**export**(*net*, **inputs*, *file_name*, *file_format*, ***kwargs*)

Export the MindSpore network into an offline model in the specified format.

Note:

1. When exporting AIR, ONNX format, the size of a single tensor can not exceed 2GB.
2. When *file_name* does not have a suffix, the system will automatically add one according to the *file_format*.
3. Exporting functions decorated with *mindspore.jit()* to mindir format is supported.
4. When exporting a function decorated with *mindspore.jit()*, the function should not involve class properties in calculations.
5. AIR format is deprecated, and will be removed in a future version, please use other format or use MindSpore Lite to do offline inference.

Parameters

- **net** (*Union[Cell, function]*) –MindSpore network.
- **inputs** (*Union[Tensor, Dataset, List, Tuple, Number, Bool]*) –It represents the inputs of the *net*, if the network has multiple inputs, set them together. While its type is Dataset, it represents the preprocess behavior of the *net*, data preprocess operations will be serialized. In second situation, you should adjust batch size of dataset script manually which will impact on the batch size of 'net' input. Only supports parse "image" column from dataset currently.

- **file_name** (*str*) –File name of the model to be exported.
- **file_format** (*str*) –MindSpore currently supports 'AIR', 'ONNX' and 'MINDIR' format for exported model.
  - AIR: Ascend Intermediate Representation. An intermediate representation format of Ascend model.
  - ONNX: Open Neural Network eXchange. An open format built to represent machine learning models.
  - MINDIR: MindSpore Native Intermediate Representation for Anf. An intermediate representation format for MindSpore models. MINDIR does not support operators which have dictionary attribute.
- **kwargs** (*dict*) –Configuration options dictionary.
  - enc_key (byte): Byte-type key used for encryption. The valid length is 16, 24, or 32.
  - enc_mode (Union[str, function]): Specifies the encryption mode, to take effect when enc_key is set.
    - For 'AIR' and 'ONNX' models, only customized encryption is supported.
    - For 'MINDIR', all options are supported. Option: 'AES-GCM', 'AES-CBC', 'SM4-CBC' or Customized encryption. Default: `'AES-GCM'`.
    - For details of using the customized encryption, please check the tutorial.
  - dataset (Dataset): Specifies the preprocessing method of the dataset, which is used to import the preprocessing of the dataset into MindIR.
  - incremental (bool): export MindIR incrementally.
  - custom_func (function): Functions for custom defined export policies. This function will be used to customize the model during network export. Currently only support for files with mindir format. The function only accepts one input representing the proto object of the mindir file. When modifying a model, it is necessary to ensure the correctness of the *custom_func* , otherwise it may lead to model loading failure or functional errors. Default: `None` .

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> input_tensor = Tensor(np.ones([1, 1, 32, 32]).astype(np.float32))
>>> ms.export(net, input_tensor, file_name='lenet', file_format='MINDIR')
>>>
>>> # Export model in MindIR format and modified the model info using custom_func
>>> # The custom_func only support one input representing the Proto object of the model
>>> # And custom_func does not support return value
>>> def _custom_func(mindir_model):
...     mindir_model.producer_name = "test11111"
...     mindir_model.producer_version = "11.0"
...     mindir_model.user_info["version"] = "11.0"
>>> ms.export(net, input_tensor, file_name="lenet", file_format='MINDIR', custom_func=_
↪custom_func)
```

Tutorial Examples:

- Saving and Loading the Model - Saving and Loading MindIR

1.5.6 mindspore.get_ckpt_path_with_strategy

mindspore.**get_ckpt_path_with_strategy**(*cur_ckpt_path*, *cur_strategy_path*)

Find available checkpoint file path from all backup checkpoint files of current rank. It suppose that checkpoint path contains substring 'rank_{rank_id}' which is used to distinguish between different path.If cur_ckpt_path doesn't have 'rank_{rank_id}' substring, will return cur_ckpt_path itself when cur_ckpt_path is exist, otherwise return None.

Note: This API must be called after the communication is initialized because the cluster information needs to be obtained internally.

Parameters

- **cur_ckpt_path** (*str*) –the checkpoint file path which cur rank needs.
- **cur_strategy_path** (*str*) –strategy file path for current rank.

Returns

- new_ckpt_file (str), if found available checkpoint file , return it.
- None, if not found available checkpoint, return None.

Examples

```
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore import get_ckpt_path_with_strategy
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.DATA_PARALLEL, gradients_
↪mean=True)
>>> init()
>>> ckpt_file= "./rank_5/iteration-1_40.ckpt"
>>> strategy_file = "./src_pipeline_strategys/src_strategy_5.ckpt"
>>> ckpt_file_new = get_ckpt_path_with_strategy(ckpt_file, strategy_file)
>>> print(ckpt_file_new)
```

1.5.7 mindspore.load

mindspore.**load**(*file_name*, ***kwargs*)

Load MindIR.

The returned object can be executed by a *GraphCell*, see class *mindspore.nn.GraphCell* for more details.

Parameters

- **file_name** (*str*) –MindIR file name.
- **kwargs** (*dict*) –Configuration options dictionary.
  - dec_key (bytes): Byte-type key used for decryption. The valid length is 16, 24, or 32.
  - dec_mode (Union[str, function], optional): Specifies the decryption mode, to take effect when dec_key is set.
    - Option: 'AES-GCM', 'AES-CBC', 'SM4-CBC' or customized decryption. Default: 'AES-GCM'.
    - For details of using the customized decryption, please check the tutorial.

Returns

GraphCell, a compiled graph that can executed by *GraphCell*.

Raises

- **NotImplementedError** –Dynamic model structure obfuscation is no longer supported.
- **ValueError** –MindIR file does not exist or *file_name* is not a string.
- **RuntimeError** –Failed to parse MindIR file.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore import context
>>> context.set_context(mode=context.GRAPH_MODE)
>>>
>>> net = nn.Conv2d(1, 1, kernel_size=3, weight_init="ones")
>>> input_tensor = Tensor(np.ones([1, 1, 3, 3]).astype(np.float32))
>>> ms.export(net, input_tensor, file_name="net", file_format="MINDIR")
>>> graph = ms.load("net.mindir")
>>> net = nn.GraphCell(graph)
>>> output = net(input_tensor)
>>> print(output)
[[[[4. 6. 4.]
   [6. 9. 6.]
   [4. 6. 4.]]]]
```

Tutorial Examples:

- Saving and Loading the Model - Saving and Loading MindIR

1.5.8 mindspore.load_checkpoint

mindspore.**load_checkpoint**(*ckpt_file_name*, *net=None*, *strict_load=False*, *filter_prefix=None*, *dec_key=None*, *dec_mode='AES-GCM'*, *specify_prefix=None*, *choice_func=None*, *crc_check=False*, *remove_redundancy=False*, *format='ckpt'*)

Load checkpoint info from a specified file.

Note:

- *specify_prefix* and *filter_prefix* are in the process of being deprecated, *choice_func* is recommended instead. *specify_prefix* and *filter_prefix* do not affect each other. And using either of those two args will override *choice_func* at the same time.
- If none of the parameters are loaded from checkpoint file, it will throw ValueError.
- When loading a checkpoint that has removed redundancy, the network should be compiled.

Parameters

- **ckpt_file_name** (*str*) –Checkpoint file name.
- **net** (*Cell, optional*) –The network where the parameters will be loaded. Default: `None` .

- **strict_load** (*bool, optional*) –Whether to strict load the parameter into net. If `False` , it will load parameter into net when parameter name's suffix in checkpoint file is the same as the parameter in the network. When the types are inconsistent perform type conversion on the parameters of the same type, such as float32 to float16. Default: `False` .
- **filter_prefix** (*Union[str, list[str], tuple[str]], optional*) –Deprecated(see *choice_func*). Parameters starting with the filter_prefix will not be loaded. Default: `None` .
- **dec_key** (*Union[None, bytes], optional*) –Byte type key used for decryption. If the value is `None` , the decryption is not required. Default: `None` .
- **dec_mode** (*str, optional*) –This parameter is valid only when dec_key is not set to `None` . Specifies the decryption mode, currently supports `"AES-GCM"` and `"AES-CBC"` and `"SM4-CBC"` . Default: `"AES-GCM"` .
- **specify_prefix** (*Union[str, list[str], tuple[str]], optional*) –Deprecated(see *choice_func*). Parameters starting with the specify_prefix will be loaded. Default: `None` .
- **choice_func** (*Union[None, function], optional*) –Input value of the function is a Parameter name of type string, and the return value is a bool. If returns `True` , the Parameter that matches the custom condition will be loaded. If returns `False` , the Parameter that matches the custom condition will be removed. Default: `None` .
- **crc_check** (*bool, optional*) –Whether to perform crc32 validation when loading checkpoint. Default: `False` .
- **remove_redundancy** (*bool, optional*) –Whether to enable loading of checkpoint saved with redundancy removal. Redundancy removal refers to eliminating redundant data in data parallelism mode. Default: `False` , means redundant-free loading is not enabled.
- **format** (*str, optional*) –Format of the input file, can be "ckpt" or "safetensors". Default: "ckpt".

Returns

Dict, key is parameter name, value is a Parameter or string. When the *append_dict* parameter of *mindspore.save_checkpoint()* and the *append_info* parameter of *mindspore.train.CheckpointConfig* are used to save the checkpoint, *append_dict* and *append_info* are dict types, and their value are string, then the return value obtained by loading checkpoint is string, and in other cases the return value is Parameter.

Raises

- **ValueError** –Checkpoint file's format is incorrect.
- **ValueError** –Parameter's dict is None after load checkpoint file.
- **TypeError** –The type of *specify_prefix* or *filter_prefix* is incorrect.

Examples

```
>>> import mindspore as ms
>>>
>>> ckpt_file_name = "./checkpoint/LeNet5-1_32.ckpt"
>>> param_dict = ms.load_checkpoint(ckpt_file_name,
...                                 choice_func=lambda x: x.startswith("conv") and not x.
 ↪startswith("conv1"))
>>> print(param_dict["conv2.weight"])
Parameter (name=conv2.weight, shape=(16, 6, 5, 5), dtype=Float32, requires_grad=True)
>>> def func(param_name):
...     whether_load = False
...     if param_name.startswith("conv"):
...         whether_load = True
...     if param_name.startswith("conv1"):
```

(continues on next page)

(continued from previous page)

```
...         whether_load = False
...     return whether_load
>>> param_dict1 = ms.load_checkpoint(ckpt_file_name, choice_func=func)
>>> print(param_dict1["conv2.weight"])
Parameter (name=conv2.weight, shape=(16, 6, 5, 5), dtype=Float32, requires_grad=True)
>>> def func(param_name):
...     whether_load = False
...     if param_name.startswith("conv1"):
...         whether_load = True
...     return whether_load
>>> param_dict2 = ms.load_checkpoint(ckpt_file_name, choice_func=func)
>>> print(param_dict2)
{'conv1.weight': Parameter (name=conv1.weight, shape=(6, 1, 5, 5), dtype=Float32, requires_
↪grad=True)}
```

Tutorial Examples:

- Saving and Loading the Model - Saving and Loading the Model Weight

1.5.9 mindspore.load_checkpoint_async

mindspore.**load_checkpoint_async**(*ckpt_file_name*, *net=None*, *strict_load=False*, *filter_prefix=None*, *dec_key=None*, *dec_mode='AES-GCM'*, *specify_prefix=None*, *choice_func=None*)

Load checkpoint info from a specified file asyncly.

> **Warning:** This is an experimental API that is subject to change or deletion.

Note:

- *specify_prefix* and *filter_prefix* do not affect each other.
- If none of the parameters are loaded from checkpoint file, it will throw ValueError.
- *specify_prefix* and *filter_prefix* are in the process of being deprecated, *choice_func* is recommended instead. And using either of those two args will override *choice_func* at the same time.

Parameters

- **ckpt_file_name** (*str*) –Checkpoint file name. The file extension must be *ckpt* or *safetensors* .
- **net** (*Cell, optional*) –The network where the parameters will be loaded. Default: `None` .
- **strict_load** (*bool, optional*) –Whether to strict load the parameter into net. If `False` , it will load parameter into net when parameter name's suffix in checkpoint file is the same as the parameter in the network. When the types are inconsistent perform type conversion on the parameters of the same type, such as float32 to float16. Default: `False` .
- **filter_prefix** (*Union[str, list[str], tuple[str]], optional*) –Deprecated(see *choice_func*). Parameters starting with the *filter_prefix* will not be loaded. Default: `None` .
- **dec_key** (*Union[None, bytes], optional*) –Byte type key used for decryption. If the value is `None` , the decryption is not required. Default: `None` .

- **dec_mode** (*str, optional*) –This parameter is valid only when dec_key is not set to `None` . Specifies the decryption mode, currently supports `"AES-GCM"` and `"AES-CBC"` and `"SM4-CBC"` . Default: `"AES-GCM"` .
- **specify_prefix** (*Union[str, list[str], tuple[str]], optional*) –Deprecated(see *choice_func*). Parameters starting with the specify_prefix will be loaded. Default: `None` .
- **choice_func** (*Union[None, function], optional*) –Input value of the function is a Parameter name of type string, and the return value is a bool. If returns `True` , the Parameter that matches the custom condition will be loaded. If returns `False` , the Parameter that matches the custom condition will be removed. Default: `None` .

Returns

A custom inner class, calling its *result* method yields the *mindspore.load_checkpoint()* result.

Raises

- **ValueError** –Checkpoint file's format is incorrect.
- **ValueError** –Parameter's dict is None after load checkpoint file.
- **TypeError** –The type of *specify_prefix* or *filter_prefix* is incorrect.

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.train import Model
>>> from mindspore.amp import FixedLossScaleManager
>>> from mindspore import context
>>> from mindspore import load_checkpoint_async
>>> from mindspore import load_param_into_net
>>> mindspore.set_device(device_target="Ascend")
>>> context.set_context(mode=context.GRAPH_MODE)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> ckpt_file = "./checkpoint/LeNet5-1_32.ckpt"
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
>>> loss_scale_manager = FixedLossScaleManager()
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics=None,
...               loss_scale_manager=loss_scale_manager)
>>> pd_future = load_checkpoint_async(ckpt_file)
>>> model.build(train_dataset=dataset, epoch=2)
>>> param_dict = pd_future.result()
>>> load_param_into_net(net, param_dict)
>>> model.train(2, dataset)
>>> print("param dict len: ", len(param_dict), flush=True)
```

1.5.10 mindspore.load_mindir

mindspore.**load_mindir**(*file_name*)

load protobuf file.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

Parameters

file_name (*str*) −File name.

Returns

ModelProto, mindir proto object.

Raises

ValueError −The file does not exist or the file name format is incorrect.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> md = ms.load_mindir("test.mindir")
```

1.5.11 mindspore.load_param_into_net

mindspore.**load_param_into_net**(*net*, *parameter_dict*, *strict_load=False*, *remove_redundancy=False*)

Load parameters into network, return parameter list that are not loaded in the network.

Note: When loading a parameter dict that has removed redundancy, the network should be compiled.

Parameters

- **net** (`Cell`) −The network where the parameters will be loaded.
- **parameter_dict** (*dict*) −The dictionary generated by load checkpoint file, it is a dictionary consisting of key: parameters's name, value: parameter.
- **strict_load** (*bool, optional*) −Whether to strict load the parameter into net. If `False` , it will load parameter into net when parameter name's suffix in checkpoint file is the same as the parameter in the network. When the types are inconsistent perform type conversion on the parameters of the same type, such as float32 to float16. Default: `False` .
- **remove_redundancy** (*bool, optional*) −Whether to enable loading of checkpoint saved with redundancy removal. Redundancy removal refers to eliminating redundant data in data parallelism mode. Default: `False` , means redundant-free loading is not enabled.

Returns

- param_not_load (List), the parameter name in model which are not loaded into the network.

- ckpt_not_load (List), the parameter name in checkpoint file which are not loaded into the network.

Raises

TypeError – Argument is not a Cell, or parameter_dict is not a Parameter dictionary.

Examples

```
>>> import mindspore as ms
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> ckpt_file_name = "./checkpoint/LeNet5-1_32.ckpt"
>>> param_dict = ms.load_checkpoint(ckpt_file_name, filter_prefix="conv1")
>>> param_not_load, _ = ms.load_param_into_net(net, param_dict)
>>> print(param_not_load)
['conv1.weight']
```

Tutorial Examples:

- Saving and Loading the Model - Saving and Loading the Model Weight

1.5.12 mindspore.parse_print

mindspore.**parse_print**(*print_file_name*)

Parse data file generated by *mindspore.ops.Print*.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

Parameters

print_file_name (*str*) – The file name needs to be parsed.

Returns

List, element of list is Tensor.

Raises

- **ValueError** – The print file does not exist or is empty.
- **RuntimeError** – Failed to parse the file.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor, ops
>>> ms.set_context(mode=ms.GRAPH_MODE, print_file_path='log.data')
>>> class PrintInputTensor(nn.Cell):
...         def __init__(self):
...             super().__init__()
...             self.print = ops.Print()
```

(continues on next page)

(continued from previous page)

```
...
...         def construct(self, input_pra):
...             self.print('print:', input_pra)
...             return input_pra
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]).astype(np.float32)
>>> input_pra = Tensor(x)
>>> net = PrintInputTensor()
>>> net(input_pra)
>>>
>>> data = ms.parse_print('./log.data')
>>> print(data)
['print:', Tensor(shape=[2, 4], dtype=Float32, value=
[[ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00,  4.00000000e+00],
[ 5.00000000e+00,  6.00000000e+00,  7.00000000e+00,  8.00000000e+00]])]
```

1.5.13 mindspore.safetensors_to_ckpt

mindspore.**safetensors_to_ckpt**(*file_path*, *save_path=None*, *name_map=None*, *file_name_regex=None*, *processes_num=1*)

Converts safetensors files into MindSpore checkpoint format and saves them to *save_path*. Safetensors is a reliable and portable machine learning model storage format introduced by Huggingface, used for securely storing Tensors with fast speed (zero copy).

Note: The number of multiprocess settings is related to the size of the host, and it is not recommended to set it too large, otherwise it may cause freezing.

Parameters

- **file_path** (*str*) –Path to the directory containing safetensors files or a single safetensors file (.safetensors).
- **save_path** (*str, optional*) –Directory path where checkpoint files will be saved. Defaults: `None`.
- **name_map** (*dict, optional*) –Dictionary mapping original parameter names to new names. Defaults: `None`.
- **file_name_regex** (*str, optional*) –Regular expression used to match the file that needs to be converted. Defaults: `None`.
- **processes_num** (*int, optional*) –Number of processes to use for parallel processing. Defaults: 1.

Raises

ValueError –If the input path is invalid, the save_path is not a directory, or the file_path does not end with '.safetensors'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> ms.safetensors_to_ckpt("./safetensors_save_path")
>>> ms.safetensors_to_ckpt("./safetensors_save_path/rank0/checkpoint_0.safetensors")
>>> ms.safetensors_to_ckpt("./safetensors_save_path/rank0/checkpoint_0.safetensors", "./new_
↪path/")
>>> namemap = {"lin.weight":"new_name"}
>>> ms.safetensors_to_ckpt("./safetensors_save_path/rank0/checkpoint_0.safetensors", "./new_
↪path/", namemap)
```

1.5.14 mindspore.save_checkpoint

mindspore.**save_checkpoint**(*save_obj*, *ckpt_file_name*, *integrated_save=True*, *async_save=False*, *append_dict=None*, *enc_key=None*, *enc_mode='AES-GCM'*, *choice_func=None*, *crc_check=False*, *format='ckpt'*, ***kwargs*)

Save checkpoint to a specified file.

Note: The *enc_mode* and *crc_check* parameters are mutually exclusive and cannot be configured simultaneously.

Parameters

- **save_obj** (*Union[Cell, list, dict]*) –The object to be saved. The data type can be *mindspore.nn.Cell*, list, or dict.
  - If a list, it can be the returned value of *Cell.trainable_params()*, or a list of dict elements(each element is a dictionary, like [{"name": param_name, "data": param_data},···], the type of *param_name* must be string, and the type of *param_data* must be parameter or Tensor).
  - If dict, it can be the returned value of *mindspore.load_checkpoint()*.
- **ckpt_file_name** (*str*) –Checkpoint file name. If the file name already exists, it will be overwritten.
- **integrated_save** (*bool*) –Whether to integrated save in automatic model parallel scene. Default: `True` .
- **async_save** (*Union[bool, str], optional*) –Whether to use asynchronous saving of the checkpoint file or safetensors file, if True, the asynchronous thread is used by default. If the type is string, the method of asynchronous saving, it can be "process" or "thread". Default: `False` .
- **append_dict** (*dict*) –Additional information that needs to be saved. The key of dict must be str, the value of dict must be one of int, float, bool, string, Parameter or Tensor. Default: `None` .
- **enc_key** (*Union[None, bytes]*) –Byte type key used for encryption. If the value is `None` , the encryption is not required. Default: `None` .
- **enc_mode** (*str*) –This parameter is valid only when enc_key is not set to `None` . Specifies the encryption mode, currently supports `"AES-GCM"` and `"AES-CBC"` and `"SM4-CBC"` . Default: `"AES-GCM"` .
- **choice_func** (*function*) –A function for saving custom selected parameters. The input value of *choice_func* is a parameter name in string type, and the returned value is a bool. Default: `None` .
  - If returns `True` , the Parameter that matching the custom condition will be saved.
  - If returns `False` , the Parameter that not matching the custom condition will not be saved.
- **crc_check** (*bool*) –Whether to perform crc32 calculation when saving checkpoint and save the calculation result to the file. Default: `False` .

- **format** (*str*) –Format of the output file, can be "ckpt" or "safetensors". Default: "ckpt".
- **kwargs** (*dict*) –Configuration options dictionary.

Raises

- **TypeError** –If the parameter *save_obj* is not *mindspore.nn.Cell* , list or dict type.
- **TypeError** –If the parameter *integrated_save* is not bool type.
- **TypeError** –If the parameter *ckpt_file_name* is not string type.
- **TypeError** –If the parameter *async_save* is not bool or string type.
- **ValueError** –If the parameter *async_save* is string type but not in ["process", "thread"].

Examples

```
>>> import mindspore as ms
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> ms.save_checkpoint(net, "./lenet.ckpt",
...                    choice_func=lambda x: x.startswith("conv") and not x.startswith("conv1
 ↪"))
>>> param_dict1 = ms.load_checkpoint("./lenet.ckpt")
>>> print(param_dict1)
{'conv2.weight': Parameter (name=conv2.weight, shape=(16, 6, 5, 5), dtype=Float32, requires_
 ↪grad=True)}
>>> params_list = net.trainable_params()
>>> ms.save_checkpoint(params_list, "./lenet_list.ckpt",
...                    choice_func=lambda x: x.startswith("conv") and not x.startswith("conv2
 ↪"))
>>> param_dict2 = ms.load_checkpoint("./lenet_list.ckpt")
>>> print(param_dict2)
{'conv1.weight': Parameter (name=conv1.weight, shape=(6, 1, 5, 5), dtype=Float32, requires_
 ↪grad=True)}
>>> ms.save_checkpoint(param_dict2, "./lenet_dict.ckpt")
>>> param_dict3 = ms.load_checkpoint("./lenet_dict.ckpt")
>>> print(param_dict3)
{'conv1.weight': Parameter (name=conv1.weight, shape=(6, 1, 5, 5), dtype=Float32, requires_
 ↪grad=True)}
```

Tutorial Examples:

- Saving and Loading the Model - Saving and Loading the Model Weight

1.5.15 mindspore.save_mindir

mindspore.**save_mindir**(*model*, *file_name*)

save protobuf file.

Note: The interface is deprecated from version 2.5 and will be removed in a future version.

Parameters

- **model** (*ModelProto*) –mindir model
- **file_name** (*str*) –File name.

Raises

- **TypeError** –The argument *model* is not a ModelProto object.
- **ValueError** –The file path does not exist or the *file_name* format is incorrect.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> md = ms.load_mindir("test.mindir")
>>> md.user_info["version"]="pangu v100"
>>> ms.save_mindir(md,"test_new.mindir")
>>> md_new = ms.load_mindir("test_new.mindir")
>>> md_new.user_info
```

1.6 Automatic Differentiation

mindspore.grad	A wrapper function to generate the gradient function for the input function.
mindspore.value_and_grad	A wrapper function to generate the function to calculate forward output and gradient for the input function.
mindspore.get_grad	When *return_ids* of *mindspore.grad()* or *mindspore.grad()* is set to True , use return value of *mindspore.grad*, or the second return value of *mindspore.grad* as gradients.
mindspore.jacfwd	Compute Jacobian via forward mode, corresponding to forward-mode differentiation.
mindspore.jacrev	Compute Jacobian via reverse mode, corresponding to reverse-mode differentiation.
mindspore.jvp	Compute the jacobian-vector-product of the given network.
mindspore.vjp	Compute the vector-jacobian-product of the given network.

1.6.1 mindspore.grad

mindspore.**grad**(*fn*, *grad_position=0*, *weights=None*, *has_aux=False*, *return_ids=False*)

A wrapper function to generate the gradient function for the input function.

As for gradient, three typical cases are included:

1. gradient with respect to inputs. In this case, *grad_position* is not None while *weights* is None.
2. gradient with respect to weights. In this case, *grad_position* is None while *weights* is not None.
3. gradient with respect to inputs and weights. In this case, *grad_position* and *weights* are not None.

Parameters

- **fn** (*Union[Cell, Function]*) –Function to do GradOperation.
- **grad_position** (*Union[NoneType, int, tuple[int]]*) –Index to specify which inputs to be differentiated. Default: `0` .
  - If int, get the gradient with respect to single input.
  - If tuple, get the gradients with respect to selected inputs. *grad_position* begins with 0.
  - If None, none derivative of any input will be figured out, and in this case, *weights* is required.
- **weights** (*Union[ParameterTuple, Parameter, list[Parameter]]*) –The parameters of the training network that need to calculate the gradient. *weights* can be got through *weights = net.trainable_params()* . Default: `None` .
- **has_aux** (*bool*) –If `True` , only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: `False` .
- **return_ids** (*bool*) –Whether return the tuple made by gradients and the index to specify which inputs to be differentiated or the name of parameters of the training network that need to calculate the gradient. If `True` , the output gradients will be replaced by the tuples made by gradients and the index to specify which inputs to be differentiated or the name of parameters of the training network. Default: `False` .

Returns

Function, the gradient function to calculate gradient for the input function or cell. For example, as for *out1, out2 = fn(*args)*, when *has_aux* is set `True` , gradient function will return outputs like *(gradient, out2)* and *out2* does not contribute to the differentiation, otherwise *gradient*. When return_ids is set to `True` , the format of the output will be the same with the output of grad when return_ids is set to `False`, but every gradient in the output will be replaced by a tuple of position id or parameter name and its gradient.

Raises

- **ValueError** –If both *grad_position* and *weights* are None.
- **TypeError** –If type of Args does not belong to required ones.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops, nn, grad
>>>
>>> # Cell object to be differentiated
>>> class Net(nn.Cell):
...     def construct(self, x, y, z):
...         return x * y * z
>>> x = Tensor([1, 2], mindspore.float32)
>>> y = Tensor([-2, 3], mindspore.float32)
>>> z = Tensor([0, 3], mindspore.float32)
>>> net = Net()
>>> output = grad(net, grad_position=(1, 2))(x, y, z)
>>> print(output)
(Tensor(shape=[2], dtype=Float32, value=[ 0.00000000e+00,  6.00000000e+00]),
 Tensor(shape=[2], dtype=Float32, value=[-2.00000000e+00,  6.00000000e+00]))
>>>
>>> # Function object to be differentiated
>>> def fn(x, y, z):
...     res = x * ops.exp(y) * ops.pow(z, 2)
...     return res, z
>>> x = Tensor([3, 3], mindspore.float32)
>>> y = Tensor([0, 0], mindspore.float32)
>>> z = Tensor([5, 5], mindspore.float32)
>>> gradient, aux = grad(fn, (1, 2), None, True)(x, y, z)
>>> print(gradient)
(Tensor(shape=[2], dtype=Float32, value= [ 7.50000000e+01,  7.50000000e+01]),
 Tensor(shape=[2], dtype=Float32, value= [ 3.00000000e+01,  3.00000000e+01]))
>>> print(aux)
(Tensor(shape=[2], dtype=Float32, value= [ 5.00000000e+00,  5.00000000e+00]),)
>>>
>>> # For given network to be differentiated with both inputs and weights, there are 4 cases.
>>> net = nn.Dense(10, 1)
>>> loss_fn = nn.MSELoss()
>>> def forward(inputs, labels):
...     logits = net(inputs)
...     loss = loss_fn(logits, labels)
...     return loss, logits
>>> inputs = Tensor(np.random.randn(16, 10).astype(np.float32))
>>> labels = Tensor(np.random.randn(16, 1).astype(np.float32))
>>> weights = net.trainable_params()
>>> # Case 1: gradient with respect to inputs.
>>> # Aux value does not contribute to the gradient.
>>> grad_fn = grad(forward, grad_position=(0, 1), weights=None, has_aux=True)
>>> inputs_gradient, (aux_logits,) = grad_fn(inputs, labels)
>>> print(len(inputs_gradient))
2
>>> print(aux_logits.shape)
(16, 1)
>>>
>>> # Case 2: gradient with respect to weights.
>>> grad_fn = grad(forward, grad_position=None, weights=weights, has_aux=True)
>>> params_gradient, (aux_logits,) = grad_fn(inputs, labels)
>>> print(len(weights), len(params_gradient))
2 2
```

(continues on next page)

(continued from previous page)

```
>>> print(aux_logits.shape)
(16, 1)
>>>
>>> # Case 3: gradient with respect to inputs and weights.
>>> grad_fn = grad(forward, grad_position=0, weights=weights, has_aux=False)
>>> inputs_gradient, params_gradient = grad_fn(inputs, labels)
>>> print(len(weights), len(params_gradient))
2 2
>>> # Case 4: return the gradient with ids.
>>> import numpy as np
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import Tensor, ops
>>> from mindspore import grad
>>>
>>> # Cell object to be differentiated
>>> class Net(nn.Cell):
...     def construct(self, x, y, z):
...         return x * y * z
>>> x = Tensor([1, 2], mindspore.float32)
>>> y = Tensor([-2, 3], mindspore.float32)
>>> z = Tensor([0, 3], mindspore.float32)
>>> net = Net()
>>> output = grad(net, grad_position=(1, 2), return_ids = True)(x, y, z)
>>> print(output)
((1, Tensor(shape=[2], dtype=Float32, value=[ 0.00000000e+00,  6.00000000e+00])),
 (2, Tensor(shape=[2], dtype=Float32, value=[-2.00000000e+00,  6.00000000e+00])))
```

1.6.2 mindspore.value_and_grad

mindspore.**value_and_grad**(*fn*, *grad_position=0*, *weights=None*, *has_aux=False*, *return_ids=False*)

A wrapper function to generate the function to calculate forward output and gradient for the input function.

As for gradient, three typical cases are included:

1. gradient with respect to inputs. In this case, *grad_position* is not None while *weights* is `None`.
2. gradient with respect to weights. In this case, *grad_position* is None while *weights* is not `None`.
3. gradient with respect to inputs and weights. In this case, *grad_position* and *weights* are not `None`.

Parameters

- **fn** (*Union[Cell, Function]*) – Function to do GradOperation.
- **grad_position** (*Union[NoneType, int, tuple[int]], optional*) – Index to specify which inputs to be differentiated. Default: `0` .
  - If int, get the gradient with respect to single input.
  - If tuple, get the gradients with respect to selected inputs. *grad_position* begins with 0.
  - If None, none derivative of any input will be solved, and in this case, *weights* is required.
- **weights** (*Union[ParameterTuple, Parameter, list[Parameter]], optional*) – The parameters of the training network that need to calculate the gradient. *weights* can be got through *weights = net.trainable_params()* . Default: `None` .

- **has_aux** (*bool, optional*) –If `True` , only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: `False` .
- **return_ids** (*bool, optional*) –Whether the returned derivation function contains *grad_position* or *weights* information. If `True`, all gradient values in the returned derivation function will be replaced with: [gradient, grad_position] or [gradient, weights]. Default: `False` .

Returns

Function, the derivative function used to compute the gradient of a given function. For example, as for *out1, out2 = fn(*args)* , gradient function will return outputs like *((out1, out2), gradient)* . When *has_aux* is set to `True`, only *out1* contributes to the differentiation. If *return_ids* is `True`, all gradient values in the returned derivation function will be replaced with: [gradient, grad_position] or [gradient, weights].

Raises

- **ValueError** –If both *grad_position* and *weights* are `None`.
- **TypeError** –If type of Args does not belong to required ones.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops, nn
>>> from mindspore import value_and_grad
>>>
>>> # Cell object to be differentiated
>>> class Net(nn.Cell):
...     def construct(self, x, y, z):
...         return x * y * z
>>> x = Tensor([1, 2], mindspore.float32)
>>> y = Tensor([-2, 3], mindspore.float32)
>>> z = Tensor([0, 3], mindspore.float32)
>>> net = Net()
>>> grad_fn = value_and_grad(net, grad_position=1)
>>> output, inputs_gradient = grad_fn(x, y, z)
>>> print(output)
[-0. 18.]
>>> print(inputs_gradient)
[0. 6.]
>>>
>>> # Function object to be differentiated
>>> def fn(x, y, z):
...     res = x * ops.exp(y) * ops.pow(z, 2)
...     return res, z
>>> x = Tensor(np.array([3, 3]).astype(np.float32))
>>> y = Tensor(np.array([0, 0]).astype(np.float32))
>>> z = Tensor(np.array([5, 5]).astype(np.float32))
>>> output, inputs_gradient = value_and_grad(fn, grad_position=(1, 2), weights=None, has_
↪aux=True)(x, y, z)
>>> print(output)
(Tensor(shape=[2], dtype=Float32, value= [ 7.50000000e+01,  7.50000000e+01]),
 Tensor(shape=[2], dtype=Float32, value= [ 5.00000000e+00,  5.00000000e+00]))
```

(continues on next page)

(continued from previous page)

```
>>> print(inputs_gradient)
(Tensor(shape=[2], dtype=Float32, value= [ 7.50000000e+01,  7.50000000e+01]),
 Tensor(shape=[2], dtype=Float32, value= [ 3.00000000e+01,  3.00000000e+01]))
>>>
>>> # For given network to be differentiated with both inputs and weights, there are 3 cases.
>>> net = nn.Dense(10, 1)
>>> loss_fn = nn.MSELoss()
>>> def forward(inputs, labels):
...     logits = net(inputs)
...     loss = loss_fn(logits, labels)
...     return loss, logits
>>> inputs = Tensor(np.random.randn(16, 10).astype(np.float32))
>>> labels = Tensor(np.random.randn(16, 1).astype(np.float32))
>>> weights = net.trainable_params()
>>>
>>> # Case 1: gradient with respect to inputs.
>>> # For has_aux is set True, only loss contributes to the gradient.
>>> grad_fn = value_and_grad(forward, grad_position=0, weights=None, has_aux=True)
>>> (loss, logits), inputs_gradient = grad_fn(inputs, labels)
>>> print(logits.shape)
(16, 1)
>>> print(inputs.shape, inputs_gradient.shape)
(16, 10) (16, 10)
>>>
>>> # Case 2: gradient with respect to weights.
>>> # For has_aux is set True, only loss contributes to the gradient.
>>> grad_fn = value_and_grad(forward, grad_position=None, weights=weights, has_aux=True)
>>> (loss, logits), params_gradient = grad_fn(inputs, labels)
>>> print(logits.shape)
(16, 1)
>>> print(len(weights), len(params_gradient))
2 2
>>>
>>> # Case 3: gradient with respect to inputs and weights.
>>> # For has_aux is set False, both loss and logits contribute to the gradient.
>>> grad_fn = value_and_grad(forward, grad_position=0, weights=weights, has_aux=False)
>>> (loss, logits), (inputs_gradient, params_gradient) = grad_fn(inputs, labels)
>>> print(logits.shape)
(16, 1)
>>> print(inputs.shape, inputs_gradient.shape)
(16, 10) (16, 10)
>>> print(len(weights), len(params_gradient))
2 2
```

1.6.3 mindspore.get_grad

mindspore.**get_grad**(*gradients*, *identifier*)

When *return_ids* of *mindspore.grad()* or *mindspore.grad()* is set to `True` , use return value of *mindspore.grad*, or the second return value of *mindspore.grad* as gradients. Then find the specific gradient from *gradients* according to *identifier* .

As for gradient, two typical cases are included:

1. *identifier* is the position of the specific tensor to get gradient.
2. *identifier* is a parameter of a network.

Parameters

- **gradients** (*Union[tuple[int, Tensor], tuple[tuple, tuple]]*) –The return value of *mindspore.grad()* when *return_ids* is set to True.
- **identifier** (*Union[int, Parameter]*) –The position number of a tensor, or a parameter that is used in *mindspore.grad()*.

Returns

The Tensor gradient value corresponding to the *identifier*.

Raises

- **RuntimeError** –If gradient value corresponding to the *identifier* is not found.
- **TypeError** –If type of Args does not belong to required ones.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> from mindspore import grad, get_grad
>>>
>>>  # Cell object to be differentiated
>>> class Net(nn.Cell):
...     def construct(self, x, y, z):
...         return x * y * z
>>> x = Tensor([1, 2], mindspore.float32)
>>> y = Tensor([-2, 3], mindspore.float32)
>>> z = Tensor([0, 3], mindspore.float32)
>>> net = Net()
>>> out_grad = grad(net, grad_position=(1, 2), return_ids=True)(x, y, z)
>>> output = get_grad(out_grad, 1)
>>> print(output)
[0. 6.]
```

1.6.4 mindspore.jacfwd

`mindspore.`**jacfwd**(*fn*, *grad_position=0*, *has_aux=False*)

Compute Jacobian via forward mode, corresponding to forward-mode differentiation. When number of outputs is much greater than that of inputs, it's better to calculate Jacobian via forward mode than reverse mode to get better performance.

Parameters

- **fn** (*Union[Cell, Function]*) –Function to do GradOperation.
- **grad_position** (*Union[int, tuple[int]], optional*) –If int, get the gradient with respect to single input. If tuple, get the gradients with respect to selected inputs. 'grad_position' begins with 0. Default: `0` .
- **has_aux** (*bool, optional*) –If `True` , only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: `False` .

Returns

Function, returns the Jacobian function for the input function or cell. For example, as for *out1, out2 = fn(*args)*, when *has_aux* is set `True` , gradient function will return outputs like *(Jacobian, out2)* and *out2* does not contribute to the differentiation, otherwise *Jacobian* .

Raises

`TypeError` – *grad_position* or *has_aux* does not belong to required types.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import jacfwd
>>> from mindspore import Tensor
>>> class MultipleInputsMultipleOutputsNet(nn.Cell):
...     def construct(self, x, y, z):
...         return x ** 2 + y ** 2 + z ** 2, x * y * z
>>> x = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> y = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> z = Tensor(np.array([[1, 1], [1, 1]]).astype(np.float32))
>>> net = MultipleInputsMultipleOutputsNet()
>>> jac, aux = jacfwd(net, grad_position=0, has_aux=True)(x, y, z)
>>> print(jac)
[[[[ 2.  0.]
   [ 0.  0.]]
  [[ 0.  4.]
   [ 0.  0.]]]
 [[[ 0.  0.]
   [ 6.  0.]]
  [[ 0.  0.]
   [ 0.  8.]]]]
>>> print(aux)
[[ 1.  4.]
 [ 9. 16.]]
```

1.6.5 mindspore.jacrev

`mindspore.`**`jacrev`**(*fn*, *grad_position=0*, *has_aux=False*)

Compute Jacobian via reverse mode, corresponding to reverse-mode differentiation. When number of inputs is much greater than that of outputs, it's better to calculate Jacobian via reverse mode than forward mode to get better performance.

Parameters

- **`fn`** (*`Union[Cell, Function]`*) – Function to do GradOperation.
- **`grad_position`** (*`Union[int, tuple[int]], optional`*) – If int, get the gradient with respect to single input. If tuple, get the gradients with respect to selected inputs. 'grad_position' begins with 0. Default: `0` .
- **`has_aux`** (*`bool, optional`*) – If `True` , only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: `False` .

Returns

Function, returns the Jacobian function for the input function or cell. For example, as for *out1, out2 = fn(*args)*, when *has_aux* is set `True` , gradient function will return outputs like *(Jacobian, out2)* and *out2* does not contribute to the differentiation, otherwise *Jacobian* .

Raises

`TypeError` – *grad_position* or *has_aux* does not belong to required types.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import jacrev
>>> from mindspore import Tensor
>>> class MultipleInputsMultipleOutputsNet(nn.Cell):
...     def construct(self, x, y, z):
...         return x ** 2 + y ** 2 + z ** 2, x * y * z
>>> x = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> y = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> z = Tensor(np.array([[1, 1], [1, 1]]).astype(np.float32))
>>> net = MultipleInputsMultipleOutputsNet()
>>> jac, aux = jacrev(net, grad_position=0, has_aux=True)(x, y, z)
>>> print(jac)
[[[[ 2.  0.]
   [ 0.  0.]]
  [[ 0.  4.]
   [ 0.  0.]]]
 [[[ 0.  0.]
   [ 6.  0.]]
  [[ 0.  0.]
   [ 0.  8.]]]]
>>> print(aux)
[[ 1.  4.]
 [ 9. 16.]]
```

1.6.6 mindspore.jvp

`mindspore.`**`jvp`**(*fn*, *inputs*, *v*, *has_aux=False*)

Compute the jacobian-vector-product of the given network. The calculation procedure of JVP can be found in forward-mode differentiation.

Parameters

- **`fn`** (*`Union[Function, Cell]`*) – The function or net that takes Tensor inputs and returns single Tensor or tuple of Tensors.
- **`inputs`** (*`Union[Tensor, tuple[Tensor], list[Tensor]]`*) – The inputs to *fn* .
- **`v`** (*`Union[Tensor, tuple[Tensor], list[Tensor]]`*) – The vector in jacobian-vector-product. The shape and type of *v* should be the same as *inputs* .
- **`has_aux`** (*`bool`*) – If `True` , only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: `False` .

Returns

- **net_output** (Union[Tensor, tuple[Tensor]]) - The output of *fn(inputs)* . Specially, when *has_aux* is set `True` , *netout* is the first output of *fn(inputs)* .
- **jvp** (Union[Tensor, tuple[Tensor]]) - The result of jacobian-vector-product.
- **aux_value** (Union[Tensor, tuple[Tensor]], optional) - Only when *has_aux* is `True` , *aux_value* will be returned. It means the second to last outputs of *fn(inputs)* . Specially, *aux_value* does not contribute to gradient.

Raises

TypeError – *inputs* or *v* does not belong to required types.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import jvp
>>> from mindspore import Tensor
>>> import mindspore.nn as nn
>>> class Net(nn.Cell):
...     def construct(self, x, y):
...         return x**3 + y
>>> x = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> y = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> v = Tensor(np.array([[1, 1], [1, 1]]).astype(np.float32))
>>> output = jvp(Net(), (x, y), (v, v))
>>> print(output[0])
[[ 2. 10.]
 [30. 68.]]
>>> print(output[1])
[[ 4. 13.]
 [28. 49.]]
>>>
>>> def fn(x, y):
...     return x ** 3 + y, y
>>> output, jvp_out, aux = jvp(fn, (x, y), (v, v), has_aux=True)
>>> print(output)
[[ 2. 10.]
 [30. 68.]]
>>> print(jvp_out)
[[ 4. 13.]
 [28. 49.]]
>>> print(aux)
[[ 1. 2.]
 [3. 4.]]
```

1.6.7 mindspore.vjp

mindspore.**vjp**(*fn*, **inputs*, *weights=None*, *has_aux=False*)

Compute the vector-jacobian-product of the given network. *vjp* matches reverse-mode differentiation.

Parameters

- **fn** (*Union[Function, Cell]*) – The function or net that takes Tensor inputs and returns single Tensor or tuple of Tensors.
- **inputs** (*Union[Tensor, tuple[Tensor], list[Tensor]]*) – The inputs to *fn* .
- **weights** (*Union[ParameterTuple, Parameter, list[Parameter]], optional*) – The parameters of the training network that need to calculate the gradient. *weights* can be got through *weights = net.trainable_params()* . Default: None .
- **has_aux** (*bool, optional*) – If True, only the first output of *fn* contributes the gradient of *fn*, while the other outputs will be returned straightly. It means the *fn* must return more than one outputs in this case. Default: False.

Returns

Forward outputs and function to calculate vjp.

- **net_output** (Union[Tensor, tuple[Tensor]]) - The output of *fn(inputs)*. Specially, when *has_aux* is set to True, *net_output* is the first output of *fn(inputs)*.
- **vjp_fn** (Function) - To calculate vector-jacobian-product. Its inputs are the vectors whose shape and type should be the same as *net_output* .
- **aux_value** (Union[Tensor, tuple[Tensor]], optional) - When *has_aux* is True, *aux_value* will be returned. It means the second to last outputs of *fn(inputs)*. Specially, *aux_value* does not contribute to gradient.

Raises

TypeError – *inputs* or *v* does not belong to required types.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import vjp
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def construct(self, x, y):
...         return x**3 + y
>>> x = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> y = Tensor(np.array([[1, 2], [3, 4]]).astype(np.float32))
>>> v = Tensor(np.array([[1, 1], [1, 1]]).astype(np.float32))
>>> outputs, vjp_fn = vjp(Net(), x, y)
>>> print(outputs)
[[ 2. 10.]
 [30. 68.]]
>>> gradient = vjp_fn(v)
>>> print(gradient)
```

(continues on next page)

(continued from previous page)

```
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 3.00000000e+00,  1.20000000e+01],
 [ 2.70000000e+01,  4.80000000e+01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 1.00000000e+00,  1.00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00]]))
>>> def fn(x, y):
...     return 2 * x + y, y ** 3
>>> outputs, vjp_fn, aux = vjp(fn, x, y, has_aux=True)
>>> gradient = vjp_fn(v)
>>> print(outputs)
[[ 3.  6.]
 [ 9. 12.]]
>>> print(aux)
[[ 1.  8.]
 [27. 64.]]
>>> print(gradient)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 2.00000000e+00,  2.00000000e+00],
 [ 2.00000000e+00,  2.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 1.00000000e+00,  1.00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00]]))
```

1.7 Parallel Optimization

1.7.1 Automatic Vectorization

mindspore.vmap	Vectorizing map (vmap) is a kind of higher-order function to map *fn* along the parameter axes.

mindspore.vmap

mindspore.**vmap**(*fn*, *in_axes=0*, *out_axes=0*)

Vectorizing map (vmap) is a kind of higher-order function to map *fn* along the parameter axes.

Vmap is pioneered by Jax and it removes the restriction of batch dimension on the operator, and provides a more convenient and unified operator expression. Moreover, it allows users to composite with other functional modules such as *mindspore.grad()*, to improve the development efficiency. In addition, the vectorizing map does not execute loops outside the function, but sinks loops into the primitive operations of the function for better performance. When combined with *Graph Kernel Fusion*, operational efficiency would be further improved.

Warning: This is an experimental API that is subject to change or deletion.

Note:

- The power of vmap comes from the implementation of VmapRules of primitives. Although we have designed a generalized rule for user custom operators, we can not guarantee that it works well for all operators, unknown exceptions may occur, please be aware the risk of use.
- When calling the random number generation methods within the scope of vmap, the same random number is generated

among vector functions each time. If you expect each vector branch to use different random numbers, you need to generate batch random numbers externally in advance and then transfer them to vmap.

Parameters

- **fn** (*Union[Cell, Function, CellList]*) – Function to be mapped along the parameter axes, which takes at least one argument and returns one or more Tensors or the type of data supported by the MindSpore Tensor. When it is a CellList, the model ensembling scenario, please make sure that the structure of each cell is the same and the number of cells is consistent with the sizes of the mapped axes (*axis_size*).
- **in_axes** (*Union[int, list, tuple]*) – Specifies which dimensions (axes) of the inputs should be mapped over. Default: `0` .
  - If *in_axes* is an integer, all arguments of *fn* are mapped over according to this axis index.
  - If *in_axes* is a tuple or list, which only composed of integers or Nones and the length should equal to the number of positional arguments to *fn*, indicates which axis to map for each corresponding positional argument. Note that, axis integers must be in range $[-ndim, ndim)$ for each argument, where *ndim* is the number of dimensions of the corresponding argument.
  - None means not mapping along any axis. Also the mapping axis index of the *in_axes* must have at least one positional parameter not None. The sizes of the mapped axes (*axis_size*) for all arguments must be equal.
- **out_axes** (*Union[int, list, tuple]*) – Specifies where the mapped dimensions (axes) should appear in the outputs. Default: `0` .
  - If *out_axes* is an integer, all outputs of *fn* are specified according to this axis.
  - If *out_axes* is a tuple or list, which only composed of integers or Nones. And its length also should be equal to the number of outputs of *fn*. Note that, axis integers must be in range $[-ndim, ndim)$ for each output, where *ndim* is the dimension of the output of the *vmap*-mapped function.
  - All outputs with a non-None mapped axis must specify a non-None *out_axes*, and if outputs with None mapped axis specifies a non-None *out_axes*, the result broadcasts across the mapped axis.

Returns

Function, returns the Vectorized/Batched version function of *fn*. The arguments and outputs of this function correspond to those of *fn*, but it adds an extra batch dimension at positions specified by *in_axes* and *out_axes*.

Raises

RuntimeError –

- If base elements in *in_axes* or *out_axes* are not a None or an integer.
- If the all base elements in *in_axes* or *out_axes* are None.
- If *in_axes* is not single integer, and the length of *in_axes* is not equal to the arguments sizes.
- If *out_axes* is not single integer, and the length of *out_axes* is not equal to the outputs sizes.
- If the *axis_size* of each arguments in the scope of *vmap* are not equal.
- If the axis in *in_axes* or *out_axes* is out of bounds.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import vmap
>>> def test_vmap(x, y, z):                                                    # ([a],[a],[a]) ->␣
↪[a]
...     return x + y + z
>>> x = Tensor(np.array([[1, 2], [3, 4], [5, 6]]).astype(np.float32))         # [b, a]
>>> y = Tensor(np.array([[-3, -2, -1], [3, 2, 1]]).astype(np.float32))        # [a, b]
>>> z = Tensor(np.array([0, 3]).astype(np.float32))                           # [a]
>>> output = vmap(test_vmap, in_axes=(0, 1, None), out_axes=1)(x, y, z)       # ([b, a],[a, b],
↪[a]) -> [a, b]
>>> print(output)
[[-2  1  4]
 [ 8  9 10]]
```

1.7.2 Recompute

mindspore.recompute	This function is used to reduce memory, when run block, rather than storing the intermediate activation computed in forward pass, we will recompute it in backward pass.

mindspore.recompute

mindspore.**recompute**(*block*, **args*, ***kwargs*)

This function is used to reduce memory, when run block, rather than storing the intermediate activation computed in forward pass, we will recompute it in backward pass.

Note:

- Recompute function only support block which inherited from Cell object.
- This function interface now only support pynative mode. you can use Cell.recompute interface in graph mode.
- When use recompute function, block object should not decorated by @jit.

Parameters

- **block** (*Cell*) – Block to be recompute.
- **args** (*tuple*) – Inputs for block object to run forward pass.
- **kwargs** (*dict*) – Optional input for recompute function.

Returns

Same as return type of block.

Raises

- **TypeError** – If *block* is not Cell object.
- **AssertionError** – If execute mode is not PYNATIVE_MODE.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import Tensor, recompute
>>> class MyCell(nn.Cell):
...     def __init__(self):
...         super(MyCell, self).__init__(auto_prefix=False)
...         self.conv = nn.Conv2d(2, 2, 2, has_bias=False, weight_init='ones')
...         self.relu = ops.ReLU()
...
...     def construct(self, x):
...         y = recompute(self.conv, x)
...         return self.relu(y)
>>> inputs = Tensor(np.ones([2, 2, 2, 2]).astype(np.float32) * 2)
>>> my_net = MyCell()
>>> grad = ops.grad(my_net)(inputs)
>>> print(grad)
[[[[2. 4.]
   [4. 8.]]
  [[2. 4.]
   [4. 8.]]]
 [[[2. 4.]
   [4. 8.]]
  [[2. 4.]
   [4. 8.]]]]
```

1.8 JIT

mindspore.JitConfig	Jit config for compile.
mindspore.jit	Create a callable MindSpore graph from a Python function.
mindspore.jit_class	Class decorator for user-defined classes.
mindspore.ms_memory_recycle	Recycle memory used by MindSpore.
mindspore.mutable	Make a constant value mutable.
mindspore.constexpr	Used to calculate constant in graph copmpiling process and improve compile performance in GRAPH_MODE.
mindspore.lazy_inline	Make the cell to be reusable.
mindspore.no_inline	Make the function to be reusable.
mindspore.set_recursion_limit	Specify the recursion depth limit of function call before compiling graph.

1.8.1 mindspore.JitConfig

class mindspore.**JitConfig**(*jit_level=''*, *exc_mode='auto'*, *jit_syntax_level=''*, *debug_level='RELEASE'*, *infer_boost='off'*, ***kwargs*)

Jit config for compile.

Parameters

- **jit_level** (*str, optional*) –Used to control the compilation optimization level. Supports ["O0", "O1", "O2"]. Default: "" , The framework automatically selects the execution method. Not recommended, it is recommended to use the JIT decorator.
  - "O0": Except for optimizations that may affect functionality, all other optimizations are turned off, adopt KernelByKernel execution mode.
  - "O1": Using commonly used optimizations and automatic operator fusion optimizations, adopt KernelByKernel execution mode.
  - "O2": Ultimate performance optimization, adopt Sink execution mode.
- **exc_mode** (*str, optional*) –Control the execution mode of the model. Supports ["auto", "sink", "no_sink"]. Default: "auto" .
  - "auto": The framework automatically selects the execution method.
  - "sink": Support the network to load and load the entire device at once, and then execute it by input driver, without the need to iterate through each operator to achieve better execution performance. This mode is only supported on the Ascend backend.
  - "no_sink": The network model is executed asynchronously one by one using a single operator.
- **jit_syntax_level** (*str, optional*) –JIT syntax level for graph compiling. The value must be "STRICT" , "LAX" or "" . Default to an empty string, which means that this JitConfig configuration will be ignored and the jit_syntax_level of ms.context will be used. For more details about ms.context, refer to set_context . Default: "" .
  - "STRICT": Only basic syntax is supported, and execution performance is optimal. Can be used for MindIR load and export.
  - "LAX": Compatible with all Python syntax as much as possible. However, execution performance may be affected and not optimal. Cannot be used for MindIR load and export due to some syntax that may not be able to be exported.
- **debug_level** (*str, optional*) –Set debugging level for graph compiling. The value must be "RELEASE" or "DEBUG". Default value: RELEASE.
  - RELEASE: Used for normally running, and some debug information will be discard to get a better compiling performance.
  - DEBUG: Used for debugging when errors occur, more information will be record in compiling process.
- **infer_boost** (*str, optional*) –enable infer boost mode. The value must be "on" , "off". Default to an "off", which means that disable infer boost. when infer boost mode is enabled, MindSpore will use high perf kernel lib, use faster runtime make infer speed is best. Note: current infer boost only support *jit_level* == "O0" and only Atlas A2 series products are supported.
- ****kwargs** (*dict*) –A dictionary of keyword arguments that the class needs.

Examples

```
>>> from mindspore import JitConfig
>>>
>>> jitconfig = JitConfig(jit_level="O1")
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>>
>>> net.set_jit_config(jitconfig)
```

1.8.2 mindspore.jit

mindspore.**jit**(*function: Optional[Callable] = None*, ***, *capture_mode: str = 'ast'*, *jit_level: str = 'O0'*, *dynamic: int = 0*, *fullgraph: bool = False*, *backend: str = ''*, ***options*)

Create a callable MindSpore graph from a Python function.

This allows the MindSpore runtime to apply optimizations based on graph.

Note:

- It is not supported to run a function with decoration @jit(capture_mode="bytecode") in static graph mode, in which case the decoration @jit(capture_mode="bytecode") is considered invalid.
- Calls to functions with decorated @jit(capture_mode="bytecode") inside functions decorated with @jit(capture_mode="ast") are not supported, and the decoration @jit(capture_mode="bytecode") is considered invalid.

Parameters

function (*Function, optional*) – The Python function that will be run as a graph. Default: None.

Keyword Arguments

- **capture_mode** (*str, optional*) – The method to create a callable MindSpore graph. The value of capture_mode should be ast , bytecode or trace . Default: ast .
  - ast : Parse Python ast to build graph.
  - bytecode : Parse Python bytecode to build graph at runtime. This is an experimental prototype that is subject to change and/or deletion.
  - *trace* : Trace the execution of Python code to build graph. This is an experimental prototype that is subject to change and/or deletion.
- **jit_level** (*str, optional*) – Used to control the compilation optimization level. Currently is only effective with default backend. The value of jit_level should be O0 or O1 . Default: O0 .
  - *O0*: Except for optimizations that may affect functionality, all other optimizations are turned off.
  - *O1*: Using commonly used optimizations and automatic operator fusion optimizations. This optimization level is experimental and is being improved.
- **dynamic** (*int, optional*) – Whether dynamic shape compilation should be performed. Default: 0. The value range is as follows:
  - *0*: Do not perform dynamic shape compilation.
  - *1*: Enable dynamic shape compilation and automatically detect shape changes.

- **fullgraph** (bool, optional) – Whether to capture the entire function into graph. If False, jit attempts to be compatible with all Python syntax in the function as much as possible. If True, we require that the entire function can be captured into graph. If this is not possible (that is, if there is Python syntax not supported), then it will raise an exception. This currently only applies when capture_mode is ast. Default: `False`.
- **backend** (str, optional) – The compilation backend to be used. If this parameter is not set, the framework will use `GE` backend for Atlas training series products and `ms_backend` backend for others including Atlas A2 training series products by default.
  - *ms_backend*: Adopt KernelByKernel execution mode.
  - *GE*: Adopt Sink execution mode. The whole model will be sinked to device to execute, only applicable to the top cell of model. And only can be used in Ascend platform.
- ****options** (dict) – A dictionary of options to pass to the compilation backend.

  Some options are device specific, see the below table for details:

Option Parameters	Hardware Platform Support	Backend Support
disable_format_transform	GPU	ms_backend
exec_order	Ascend	ms_backend
ge_options	Ascend	GE
infer_boost	Ascend	ms_backend

  - disable_format_transform (bool, optional): Whether to disable the automatic format transform function from NCHW to NHWC. When the network training performance of fp16 is worse than fp32, *disable_format_transform* can be set to `True` to try to improve training performance. Default: `False` .
  - exec_order (str, optional): Set the sorting method for operator execution, currently only two sorting methods are supported: `bfs` and `dfs` . Default: `bfs` .
    - *bfs*: The default sorting method, breadth priority, good communication masking, relatively good performance.
    - *dfs*: An optional sorting method, depth-first sorting. The performance is relatively worse than that of bfs execution order, but it occupies less memory. It is recommended to try dfs in scenarios where other execution orders run out of memory (OOM).
  - ge_options (dict): Set options for ge backend. The options are divided into two categories: global, and session. This is an experimental prototype that is subject to change and/or deletion. For detailed information, please refer to Ascend community .
    - global (dict): Set global options.
    - session (dict): Set session options.
  - infer_boost (str, optional): Used to control the inference mode. Default: `off`, which means the inference mode is disabled. The range is as follows:
    - *on*: Enable inference mode, get better infer performance.
    - *off*: Disable inference mode, use forward for inference. The performance is poor.

Returns

Function, if *fn* is not None, returns a callable function that will execute the compiled function; If *fn* is None, returns a decorator and when this decorator invokes with a single *fn* argument, the callable function is equal to the case when *fn* is not None.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> from mindspore import jit
...
>>> x = Tensor(np.ones([1, 1, 3, 3]).astype(np.float32))
>>> y = Tensor(np.ones([1, 1, 3, 3]).astype(np.float32))
...
>>> # create a callable MindSpore graph by calling jit
>>> def tensor_add(x, y):
...     z = x + y
...     return z
...
>>> tensor_add_graph = jit(function=tensor_add)
>>> out = tensor_add_graph(x, y)
...
>>> # create a callable MindSpore graph through decorator @jit
>>> @jit
... def tensor_add_with_dec(x, y):
...     z = x + y
...     return z
...
>>> out = tensor_add_with_dec(x, y)
...
>>> # create a callable MindSpore graph and capture the entire function into the graph
>>> @jit(fullgraph=True)
... def tensor_add_fullgraph(x, y):
...     z = x + y
...     return z
...
>>> out = tensor_add_fullgraph(x, y)
```

1.8.3 mindspore.jit_class

mindspore.**jit_class**(*cls*)

Class decorator for user-defined classes.

This allows MindSpore to identify user-defined classes and thus obtain their attributes and methods.

Parameters

cls (*Class*) – User-defined class.

Returns

Class.

Raises

- **TypeError** – If *jit_class* is used for non-class types or nn.Cell.
- **AttributeError** – If the private attributes or magic methods of the class decorated with *jit_class* is called.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore import jit_class
...
>>> @jit_class
... class UserDefinedNet:
...     def __init__(self):
...         self.value = 10
...
...     def func(self, x):
...         return 2 * x
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.net = UserDefinedNet()
...
...     def construct(self, x):
...         out = self.net.value + self.net.func(x)
...         return out
...
>>> net = Net()
>>> out = net(5)
>>> print(out)
20
```

1.8.4 mindspore.ms_memory_recycle

mindspore.**ms_memory_recycle**()

Recycle memory used by MindSpore. When train multi Neural network models in one process, memory used by MindSpore is very large, this is because MindSpore cached runtime memory for every model. To recycle these cached memory, users can call this function after training of one model.

Examples

```
>>> import mindspore as ms
>>> ms.ms_memory_recycle()
```

1.8.5 mindspore.mutable

mindspore.**mutable**(*input_data*, *dynamic_len=False*)

Make a constant value mutable.

Currently, all the inputs of Cell except Tensor such as scalar, tuple, list and dict, are regarded as constant values. The constant values are non-differentiable and used to do constant folding in the optimization process.

Besides, currently when the network input is tuple[Tensor], list[Tensor] or dict[Tensor], even without changing the shape and dtype of the Tensors, the network will be re-compiled when calling this network repeatedly because the these inputs are regarded as constant values.

To solve the above problems, we provide api *mutable* to make the constant inputs of Cell 'mutable'. A 'mutable' input means that it is changed to be a variable input just like Tensor and the most important thing is that it will be differentiable.

When the *input_data* is tuple or list and *dynamic_len* is False, *mutable* will return a constant length tuple or list with all mutable elements. If *dynamic_len* is True, the length of the return tuple or list will be dynamic.

When a dynamic-length tuple or list returned by *mutable* is used as input to a network and the network is called repeatedly, and the length of the tuple or list is different for each run, it does not need to be re-compiled.

Parameters

- **input_data** (*Union[Tensor, scalar, tuple, list, dict]*) – The input data to be made mutable. If 'input_data' is list/tuple/dict, the type of each element should also in the valid types.
- **dynamic_len** (*bool, optional*) – Whether to set the whole sequence to be dynamic length. In graph compilation, if *dynamic_len* is `True` , the *input_data* must be list or tuple and the elements of *input_data* must have the same type and shape. Default: `False` .

Warning: This is an experimental API that is subject to change or deletion. *dynamic_len* is an experimental argument. Currently, *dynamic_len* is not supported to be `True` .

Note: Currently this api only works in GRAPH mode.

Returns

The origin input data which has been set mutable.

Raises

- **TypeError** – If *input_data* is not one of Tensor, scalar, tuple, list, dict or their nested structure.
- **TypeError** – If *dynamic_len* is `True` and *input_data* is not tuple or list.
- **ValueError** – If *dynamic_len* is `True` , *input_data* is tuple or list but the elements within *input_data* do not have the same type.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import mutable, nn, ops, Tensor, context
>>> from mindspore import dtype as mstype
>>> context.set_context(mode=context.GRAPH_MODE)
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.matmul = ops.MatMul()
...
...     def construct(self, z):
...         x = z[0]
...         y = z[1]
...         out = self.matmul(x, y)
...         return out
...
>>> class GradNetWrtX(nn.Cell):
...     def __init__(self, net):
```

(continues on next page)

(continued from previous page)

```
...         super(GradNetWrtX, self).__init__()
...         self.net = net
...         self.grad_op = ops.GradOperation()
...
...     def construct(self, z):
...         gradient_function = self.grad_op(self.net)
...         return gradient_function(z)
...
>>> z = mutable((Tensor([[0.5, 0.6, 0.4], [1.2, 1.3, 1.1]], dtype=mstype.float32),
...               Tensor([[0.01, 0.3, 1.1], [0.1, 0.2, 1.3], [2.1, 1.2, 3.3]], dtype=mstype.
↪float32)))
>>> output = GradNetWrtX(Net())(z)
>>> print(output)
(Tensor(shape=[2, 3], dtype=Float32, value=
[[ 1.41000009e+00,  1.60000002e+00,  6.59999943e+00],
 [ 1.41000009e+00,  1.60000002e+00,  6.59999943e+00]]), Tensor(shape=[3, 3], dtype=Float32,␣
↪value=
[[ 1.70000005e+00,  1.70000005e+00,  1.70000005e+00],
 [ 1.89999998e+00,  1.89999998e+00,  1.89999998e+00],
 [ 1.50000000e+00,  1.50000000e+00,  1.50000000e+00]]))
```

1.8.6 mindspore.constexpr

mindspore.**constexpr**(*fn=None*, *get_instance=True*, *name=None*, *reuse_result=True*, *check=True*)

Used to calculate constant in graph copmpiling process and improve compile performance in GRAPH_MODE.

Parameters

- **fn** (*function, optional*) – A *fn* use as the infer_value of the output operator. Default: `None` .
- **get_instance** (*bool, optional*) – If `True` , return the instance of operator, otherwise return the operator class. Default: `True` .
- **name** (*str, optional*) – Defines the operator name. If *name* is `None` , use the function name as op name. Default: `None` .
- **reuse_result** (*bool, optional*) – If `True` , the operator will be executed once and reuse the result next time, otherwise the operator will always be executed. Default: `True` .
- **check** (*bool, optional*) – If `True` , the parameters will be checked and the warning message will raised if the parameter is not const value. Default: `True` .

Examples

```
>>> import mindspore as ms
>>> # define a constant calculate function with for loop inside and use use constexpr to␣
↪accelerate the compile
>>> # process.
>>> @ms.constexpr
... def for_loop_calculate(range_num):
...     out = 0
...     for i in range(range_num):
...         if i %2 == 0 and i % 7 != 0:
...             out = out + i
...     return out // range_num
```

(continues on next page)

(continued from previous page)

```
...
>>> # construct a net and run with GRAPH_MODE.
>>> @ms.jit
... def my_func(x):
...     new_shape = for_loop_calculate(100000)
...     return ms.ops.broadcast_to(x, (new_shape, ))
...
>>> out = my_func(ms.Tensor([1]))
>>> print(out.shape)
>>> (21428, )
```

1.8.7 mindspore.lazy_inline

mindspore.**lazy_inline**(*fn=None*, *attrs=None*, *policy=None*)

Make the cell to be reusable. The corresponding sub graph will not be inline at first and will be inline with the policy. Registering the decorator of the built-in function *__init__* of a cell, the decorator will add the parameters of *__init__* according to the *attrs* as the attributes of this cell.

For a detailed description of the function, see Using the lazy_inline decorator .

Warning: This feature is only supported on Ascend and is not supported on other hardwares. The construct parameters must be positional or key word arguments and have not default values. The cell has not switch sub graph.

Parameters

- **fn** (*function*) – *__init__* function of a cell.
- **attrs** (*Union[list[string], string]*) – The attributes list to add for the cell.
- **policy** (*Union[None, "front"], optional*) – The policy of inline. Default is `None`.
  - `None`: The cell will be compiled to sub graph and will not be inline.
  - `"front"`: The cell will be compiled to sub graph first and will be inline at front end.

Returns

function, original function.

Supported Platforms:

`Ascend`

Examples

```
>>> import os
>>> import numpy as np
>>> from mindspore import Tensor
>>> import mindspore.nn as nn
>>> from mindspore import lazy_inline
>>> from mindspore import context
>>> from mindspore import ops
>>> def conv3x3(in_channels, out_channels, stride=1, padding=1, pad_mode='pad'):
...     return nn.Conv2d(in_channels, out_channels,
```

(continues on next page)

(continued from previous page)

```
...                     kernel_size=3, stride=stride, padding=padding, pad_mode=pad_mode)
...
>>> def conv1x1(in_channels, out_channels, stride=1, padding=0, pad_mode='pad'):
...     return nn.Conv2d(in_channels, out_channels,
...                      kernel_size=1, stride=stride, padding=padding, pad_mode=pad_mode)
...
>>> class Block(nn.Cell):
...     expansion = 4
...
...     @lazy_inline
...     def __init__(self,
...                  in_channels,
...                  out_channels,
...                  stride=1,
...                  down_sample=False):
...         super(Block, self).__init__()
...
...         out_chls = out_channels
...         self.conv1 = conv1x1(in_channels, out_chls, stride=1, padding=0)
...         self.bn1 = nn.BatchNorm2d(out_chls)
...
...         self.conv2 = conv3x3(out_chls, out_chls, stride=stride, padding=1)
...         self.bn2 = nn.BatchNorm2d(out_chls)
...
...         self.conv3 = conv1x1(out_chls, out_channels, stride=1, padding=0)
...         self.bn3 = nn.BatchNorm2d(out_channels)
...
...         self.relu = nn.ReLU()
...         self.downsample = down_sample
...
...         self.conv_down_sample = conv1x1(in_channels, out_channels,
...                                         stride=stride, padding=0)
...         self.bn_down_sample = nn.BatchNorm2d(out_channels)
...         self.add = ops.Add()
...
...     def construct(self, x):
...         identity = x
...
...         out = self.conv1(x)
...         out = self.bn1(out)
...         out = self.relu(out)
...
...         out = self.conv2(out)
...         out = self.bn2(out)
...         out = self.relu(out)
...
...         out = self.conv3(out)
...         out = self.bn3(out)
...
...         if self.downsample:
...             identity = self.conv_down_sample(identity)
...             identity = self.bn_down_sample(identity)
...
...         out = self.add(out, identity)
...         out = self.relu(out)
...
...         return out
```

(continues on next page)

(continued from previous page)

```
...
>>> class Net(nn.Cell):
...     def __init__(self, block, num_classes=100):
...         super(Net, self).__init__()
...
...         self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, pad_mode='pad')
...         self.bn1 = nn.BatchNorm2d(64)
...         self.relu = nn.ReLU()
...         self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, pad_mode='valid')
...
...         self.layer = self.MakeLayer(
...             block, 50, in_channels=64, out_channels=2048, stride=2)
...         self.avgpool = nn.AvgPool2d(7, 1)
...         self.flatten = ops.Flatten()
...
...     def MakeLayer(self, block, layer_num, in_channels, out_channels, stride):
...         layers = []
...         resblk = block(in_channels, out_channels,
...                        stride=stride, down_sample=True)
...         layers.append(resblk)
...
...         for _ in range(1, layer_num):
...             resblk = block(out_channels, out_channels, stride=1)
...             layers.append(resblk)
...
...         return nn.SequentialCell(layers)
...
...     def construct(self, x):
...         x = self.conv1(x)
...         x = self.bn1(x)
...         x = self.relu(x)
...         x = self.maxpool(x)
...         x = self.layer(x)
...         x = self.avgpool(x)
...         x = self.flatten(x)
...         return x
...
>>> def test_compile():
...     net = Net(Block)
...     inp = Tensor(np.ones([1, 3, 224, 224]).astype(np.float32))
...     net(inp)
...
>>> context.set_context(mode=context.GRAPH_MODE)
>>> os.environ["MS_DEV_SAVE_GRAPHS"] = "2"
>>> os.environ["MS_DEV_SAVE_GRAPHS_PATH"] = os.path.realpath("./lazy")
...
>>> test_compile()
```

1.8.8 mindspore.no_inline

mindspore.**no_inline**(*fn=None*)

Make the function to be reusable. The corresponding sub graph will not be inline.

Parameters

fn (*function*) – It is the python function. If it is a methon of a cell, please refer to *mindspore.lazy_inline()*.

Returns

function, original function.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import no_inline, Tensor, jit
>>> @no_inline
... def no_inline_fun(val):
...     x = val * 3 + 2
...     return x
>>> @jit
... def call_no_inline_fun(val):
...     for _ in range(100):
...         val = no_inline_fun(val)
...     return val
>>> call_no_inline_fun(Tensor(10))
```

1.8.9 mindspore.set_recursion_limit

mindspore.**set_recursion_limit**(*recursion_limit=1000*)

Specify the recursion depth limit of function call before compiling graph. It needs to be call when the nested function call is too deep or the number of sub graphs is too large. If recursion_limit is set larger than before, the system max stack depth should be set larger too, otherwise a *core dumped* exception may be raised because of system stack overflow.

Parameters

recursion_limit (*int, optional*) – The recursion depth limit. Must be a positive integer. Default: `1000` .

Examples

```
>>> import mindspore as ms
>>> ms.set_recursion_limit(10000)
```

1.9 Tool

1.9.1 Dataset Helper

mindspore.DatasetHelper	DatasetHelper is a class to process the MindData dataset and provides the information of dataset.
mindspore.Symbol	Symbol is a data structure to indicate the symbolic info of shape.
mindspore.connect _network_with_dataset	Connect the *network* with dataset in *dataset_helper*.
mindspore.data_sink	A wrapper function to generate a function for the input function.

mindspore.DatasetHelper

class mindspore.**DatasetHelper**(*dataset*, *dataset_sink_mode=True*, *sink_size=- 1*, *epoch_num=1*)

DatasetHelper is a class to process the MindData dataset and provides the information of dataset.

According to different contexts, change the iterations of dataset and use the same iteration for loop in different contexts.

Note: The iteration of DatasetHelper will provide one epoch data.

Parameters

- **dataset** (*Dataset*) –The dataset iterator. The dataset can be generated by dataset generator API in *mindspore.dataset* module, such as `mindspore.dataset.ImageFolderDataset`.
- **dataset_sink_mode** (*bool*) –If the value is True, GetNext is employed to fetch the data at device through the dataset pipeline, otherwise fetch the data at host by iterating through the dataset. Default: `True`.
- **sink_size** (*int*) –Control the amount of data in each sink. Must be -1 or positive. If sink_size=-1, sink the complete dataset for each epoch. If sink_size>0, sink sink_size data for each epoch. Default: `-1`.
- **epoch_num** (*int*) –The number of passes of the entire dataset to be sent. Default: `1`.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> from mindspore import dataset as ds
>>>
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> set_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=False)
>>>
>>> net = nn.Dense(10, 5)
>>> # Object of DatasetHelper is iterable
>>> for next_element in set_helper:
...     # `next_element` includes data and label, using data to run the net
...     data = next_element[0]
...     result = net(data)
```

continue_send()

Continue to send data to device at the beginning of epoch.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> from mindspore import dataset as ds
>>>
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>> dataset_helper.continue_send()
```

release()
Free up resources about data sink.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> from mindspore import dataset as ds
>>>
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>> dataset_helper.release()
```

sink_size()
Get sink_size for each iteration.

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>>
>>> # Define a dataset pipeline
>>> def generator():
...     for i in range(5):
...         yield (np.ones((32, 10)),)
>>>
>>> train_dataset = ms.dataset.GeneratorDataset(generator, ["data"])
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True, sink_size=-
 ↪1)
>>>
>>> # if sink_size==-1, then will return the full size of source dataset.
>>> sink_size = dataset_helper.sink_size()
```

stop_send()
Stop send data about data sink.

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> # Define a dataset pipeline
>>> def generator():
...     for i in range(5):
...         yield (np.ones((32, 10)),)
>>> train_dataset = ms.dataset.GeneratorDataset(generator, ["data"])
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True, sink_size=-
↪1)
>>> dataset_helper.stop_send()
```

types_shapes()
Get the types and shapes from dataset on the current configuration.

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>>
>>> # Define a dataset pipeline
>>> def generator():
...     for i in range(5):
...         yield (np.ones((32, 10)),)
>>>
>>> train_dataset = ms.dataset.GeneratorDataset(generator, ["data"])
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>>
>>> types, shapes = dataset_helper.types_shapes()
```

mindspore.Symbol

class mindspore.**Symbol**(*max=0*, *min=1*, *divisor=1*, *remainder=0*, *unique=False*, ***kawgs*)
Symbol is a data structure to indicate the symbolic info of shape.

For dynamic shape networks, compared with only setting the unknown dimensions (`None`) in *Tensor* , providing more symbolic shape info can help the framework better optimize the computation graph, to improve the performance of network execution.

Parameters

- **max** (*int*) −The maximum length of this dimension, which is valid when it's greater than *min*. Default: `0` .
- **min** (*int*) −The minimum length of this dimension. Default: `1` .
- **divisor** (*int*) −The divisor(d). When *remainder* is 0, it means this dimension can be divided by d . Default: `1` .
- **remainder** (*int*) −The remainder(r) when symbol is represented by $d * N + r, N \geq 1$. Default: `0` .
- **unique** (*bool*) −When the symbol object is used multiple times, if *unique* is `True` , the shape items of this symbol are considered to be same length, otherwise only symbol info is shared by multiple dimensions. Default: `False` .

Outputs:

Symbol.

Raises

- **TypeError** –If *max*, *min*, *divisor*, *remainder* is not an int.
- **TypeError** –If *unique* is not a bool.
- **ValueError** –If *min* is not positive value.
- **ValueError** –If *divisor* is not positive value.
- **ValueError** –If *remainder* is not in the range $[0, d)$.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor, Symbol
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.abs = ms.ops.Abs()
...     def construct(self, x):
...         return self.abs(x)
...
>>> net = Net()
>>> s1 = Symbol(divisor=8, remainder=1)
>>> s2 = Symbol(max=32, unique=True)
>>> dyn_t = Tensor(shape=(None, s1, s1, s2, s2), dtype=ms.float32)
>>> net.set_inputs(dyn_t)
>>> # the shape values of last two dimensions must be equal, because "s2" is set to "unique"
>>> net(Tensor(np.random.randn(1, 9, 17, 32, 32), dtype=ms.float32)).shape
(1, 9, 17, 32, 32)
>>> net(Tensor(np.random.randn(8, 25, 9, 30, 30), dtype=ms.float32)).shape
(8, 25, 9, 30, 30)
```

mindspore.connect_network_with_dataset

mindspore.**connect_network_with_dataset**(*network*, *dataset_helper*)

Connect the *network* with dataset in *dataset_helper*. Only supported in sink mode, (dataset_sink_mode=True).

Parameters

- **network** (Cell) –The training network for dataset.
- **dataset_helper** (DatasetHelper) –A class to process the MindData dataset, it provides the type, shape and queue name of the dataset.

Returns

Cell, a new network containing the type, shape and queue name of the dataset info.

Raises

RuntimeError –If the API was not called in dataset sink mode.

Supported Platforms:

Ascend GPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> from mindspore import dataset as ds
>>>
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> dataset_helper = ms.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>> net = nn.Dense(10, 5)
>>> net_with_dataset = ms.connect_network_with_dataset(net, dataset_helper)
```

mindspore.data_sink

mindspore.**data_sink**(*fn*, *dataset*, *sink_size=1*, *jit_config=None*, *input_signature=None*)

A wrapper function to generate a function for the input function.

Note: When using data sinking, the dataset will be automatically looped to the device. The device side can cache up to 100 batches of data and occupy no more than 2GB of memory. At this time, only the number of steps for each sinking *sink_size* needs to be considered. *sink_size* defaults to 1, indicating that each epoch only takes one batch of data from the cache for training and outputs a loss. If *sink_size* is greater than 1, each epoch takes out *sink_size* batches of data from the cache for training and outputs a loss.

Parameters

- **fn** (*Function*) –The Python function that will be run with dataset.
- **dataset** (*Dataset*) –The dataset iterator. The dataset can be generated by dataset generator API in mindspore.dataset, such as mindspore.dataset.ImageFolderDataset.
- **sink_size** (*int*) –Control the amount of data in each sink. *sink_size* must be positive integer. Default: 1 .
- **jit_config** (JitConfig) –Controls the execution mode(Graph mode/PyNative mode) of the generated function, and Jit config for compile. Default: None , means running in PyNative mode.
- **input_signature** (*Union[Tensor, List or Tuple of Tensors]*) –The Tensor which describes the input arguments. The shape and dtype of the Tensor will be supplied to this function. If input_signature is specified, each input to *fn* must be a *Tensor*. And the input parameters of *fn* cannot accept ***kwargs*. The shape and dtype of actual inputs should keep the same as *input_signature*. Otherwise, TypeError will be raised. Default: None .

Returns

Function, the generated function will be executed in data sinking mode.

Raises

ValueError –If *sink_size* is not positive integer.

Supported Platforms:

Ascend GPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import dataset as ds
>>>
>>> data = {"x": np.ones((1,), dtype=np.int32), "y": np.ones((1,), dtype=np.int32)}
>>> dataset = ds.NumpySlicesDataset(data=data)
>>>
>>> def func_net(x, y):
...     out = x + y
...     return out
>>>
>>> sink_process = ms.data_sink(func_net, dataset, sink_size=1)
>>> for _ in range(2):
...     out = sink_process()
...     print(out)
2
2
```

1.9.2 Debugging and Tuning

mindspore.profiler.profile	This class to enable the profiling of MindSpore neural networks.
mindspore.profiler._ExperimentalConfig	The purpose of this class is to configure scalable parameters when using profiles for model performance data acquisition.
mindspore.profiler.mstx	Mstx class provides profiling tools for marking and tracing on NPU.
mindspore.profiler.DynamicProfilerMonitor	This class to enable the dynamic profile monitoring of MindSpore neural networks.
mindspore.profiler.schedule	This class use to get the actions of each step.
mindspore.profiler.tensorboard_trace_handler	For each step in dynamic graph mode, call this method for online analyse.
mindspore.profiler.profiler.analyse	Analyze training performance data offline, which is invoked after performance data collection is completed.
mindspore.SummaryCollector	SummaryCollector can help you to collect some common information, such as loss, learning late, computational graph and so on.
mindspore.SummaryLandscape	SummaryLandscape can help you to collect loss landscape information.
mindspore.SummaryRecord	SummaryRecord is used to record the summary data and lineage data.
mindspore.set_dump	Enable or disable dump for the *target* and its contents.
mindspore.Profiler	The current interface is deprecated, please use: *mindspore.profiler.profile* instead.

mindspore.profiler.profile

class mindspore.profiler.**profile**(*activities: list = None*, *with_stack: bool = False*, *profile_memory: bool = False*, *data_process: bool = False*, *parallel_strategy: bool = False*, *start_profile: bool = True*, *hbm_ddr: bool = False*, *pcie: bool = False*, *sync_enable: bool = True*, *schedule: Schedule = None*, *on_trace_ready: Optional[Callable[..., Any]] = None*, *experimental_config: Optional[_ExperimentalConfig] = None*)

This class to enable the profiling of MindSpore neural networks. MindSpore users can import the mindspore.profiler.profile, initialize the profile object to start profiling, Use profile.start() to start the analysis, and use profile.stop() to stop collecting and analyzing the results. Users can visualize the results using the MindStudio Insight tool. Now, profile supports AICORE operator, AICPU operator, HostCPU operator, memory, correspondence, cluster, etc data analysis.

Parameters

- **start_profile** (*bool, optional*) –The start_profile parameter controls whether to enable or disable performance data collection based on conditions. Default: `True` .
- **activities** (*list, optional*) –The activities to collect. Default: `[ProfilerActivity.CPU, ProfilerActivity.NPU]`.
  - ProfilerActivity.CPU: Collect MindSpore framework data.
  - ProfilerActivity.NPU: Collect CANN software stack and NPU data.
  - ProfilerActivity.GPU: Collect GPU data.
- **schedule** (*schedule, optional*) –Sets the action strategy for the capture, defined by the schedule class, to be used with the step interface. Default: `None`. Performance data of all steps is collected. For details, see *mindspore.profiler.schedule* .
- **on_trace_ready** (*Callable, optional*) –Sets the callback function to be executed when the performance data is collected. Default: `None`. It indicates that only performance data is collected, but not resolved. For details, see *mindspore.profiler.tensorboard _trace_handler()* .
- **profile_memory** (*bool, optional*) –(Ascend only) Whether to collect tensor memory data, collect when `True` . When using this parameter, *activities* must set to `[ProfilerActivity.CPU, ProfilerActivity.NPU]`. Collecting operator memory data when the graph compilation level is O2 requires collecting from the first step. Default: `False` . The operator name currently collected by this parameter is incomplete. This issue will be resolved in later versions. It is recommended to use the environment variable `MS_ALLOC_CONF` instead.
- **with_stack** (*bool, optional*) –(Ascend only) Whether to collect frame host call stack data on the Python side. This data is presented in the form of a flame graph in the timeline. When using this parameter, *activities* must include `ProfilerActivity.CPU`. Default value: `False` .
- **hbm_ddr** (*bool, optional*) –(Ascend only) Whether to collect On-Chip Memory/DDR read and write rate data, collect when True. Default: `False` .
- **pcie** (*bool, optional*) –(Ascend only) Whether to collect PCIe bandwidth data, collect when True. Default: `False` .
- **data_process** (*bool, optional*) –(Ascend/GPU) Whether to collect data to prepare performance data. Default value: `False` .
- **parallel_strategy** (*bool, optional*) –(Ascend only) Whether to collect parallel policy performance data. Default value: `False` .
- **sync_enable** (*bool, optional*) –(GPU only) Whether the profiler collects operators in a synchronous way. Default: `True` .
  - True: The synchronous way. Before sending the operator to the GPU, the CPU records the start timestamp. Then the operator is returned to the CPU after execution, and the end timestamp is recorded, The duration of the operator is the difference between the two timestamps.

  - False: The asynchronous way. The duration of the operator is that of sending from the CPU to the GPU. This method can reduce the impact of adding profiler on overall training time.

- **experimental_config** (*_ExperimentalConfig, optional*) – expandable parameters can be configured in this configuration item. For details, see *mindspore.profiler._ExperimentalConfig*.

Raises

RuntimeError – When the version of CANN does not match the version of MindSpore, MindSpore cannot parse the generated ascend_job_id directory structure.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, context
>>> import mindspore.dataset as ds
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics, ExportType
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = mindspore.train.Model(net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
```

(continues on next page)

(continued from previous page)

```
...     with mindspore.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.
↪NPU],
...                                     schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                          repeat=1, skip_first=2),
...                                     on_trace_ready=mindspore.profiler.
...                                          tensorboard_trace_handler("./data"),
...                                     profile_memory=False,
...                                     experimental_config=experimental_config) as prof:
...
...         # Train Model
...         for step in range(steps):
...             train(net)
...             prof.step()
```

add_metadata(*key: str*, *value: str*)

Report custom metadata key-value pair data.

Parameters

- **key** (*str*) –The key to the metadata.
- **value** (*str*) –The value to the metadata.

Examples

```
>>> import mindspore
>>> # Profiler init.
>>> with mindspore.profiler.profile() as prof:
...     # Call Profiler add_metadata
...     prof.add_metadata("test_key", "test_value")
```

add_metadata_json(*key: str*, *value: str*)

Report custom metadata key-value pair data with the value as a JSON string data.

Parameters

- **key** (*str*) –The key to the metadata.
- **value** (*str*) –The json str format value to the metadata.

Examples

```
>>> import json
>>> import mindspore
>>> # Profiler init.
>>> with mindspore.profiler.profile() as prof:
...     # Call Profiler add_metadata_json
...     prof.add_metadata_json("test_key", json.dumps({"key1": 1, "key2": 2}))
```

start()

Turn on profile data collection. profile can be turned on by condition.

Raises

- **RuntimeError** –If the profile has already started.

- **RuntimeError** – If the *start_profile* parameter is not set or is set to `True`.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, context
>>> import mindspore.dataset as ds
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics,␣
↪ExportType
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = mindspore.train.Model(net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
...     prof = mindspore.profiler.profile(activities=[ProfilerActivity.CPU,␣
↪ProfilerActivity.NPU],
...                                       schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                           repeat=1, skip_first=2),
...                                       on_trace_ready=mindspore.profiler.
...                                           tensorboard_trace_handler("./data"),
...                                       profile_memory=False,
...                                       experimental_config=experimental_config)
...     prof.start()
...     # Train Model
```

(continues on next page)

(continued from previous page)

```
...     for step in range(steps):
...         train(net)
...         prof.step()
...     prof.stop()
```

step()

Used for Ascend, distinguish step collection and parsing performance data through schedule and on_trace_ready.

Raises

- **RuntimeError** – If the *start_profile* parameter is not set or the Profiler is not started.
- **RuntimeError** – If the *schedule* parameter is not set.

Examples: >>> import numpy as np >>> import mindspore >>> from mindspore import nn, context >>> import mindspore.dataset as ds >>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics, ExportType >>> >>> class Net(nn.Cell): ···def __init__(self): ···super(Net, self).__init__() ···self.fc = nn.Dense(2,2) ···def construct(self, x): ···return self.fc(x) >>> >>> def generator(): ···for i in range(2): ···yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32) >>> >>> def train(net): ···optimizer = nn.Momentum(net.trainable_params(), 1, 0.9) ··· loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True) ···data = ds.GeneratorDataset(generator, ["data", "label"]) ··· model = mindspore.train.Model(net, loss, optimizer) ···model.train(1, data) >>> >>> if __name__ == '__main__': ···# If the device_target is GPU, set the device_target to "GPU" ···context.set_context(mode=mindspore.GRAPH_MODE) ···mindspore.set_device("Ascend") ·····# Init Profiler ···experimental_config = mindspore.profiler._ExperimentalConfig(···profiler_level=ProfilerLevel.Level0, ···aic_metrics=AicoreMetrics.AiCoreNone, ···l2_cache=False, ···mstx=False, ···data_simplification=False, ···export_type=[ExportType.Text]) ···steps = 10 ···net = Net() ···# Note that the Profiler should be initialized before model.train ···with mindspore.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.NPU], ···schedule=mindspore.profiler.schedule(wait=1, warmup=1, active=2, ···repeat=1, skip_first=2), ···on_trace_ready=mindspore.profiler.tensorboard_trace_handler("./data"), ···profile_memory=False, ···experimental_config=experimental_config) as prof: ·····# Train Model ···for step in range(steps): ···train(net) ···prof.step()

stop()

Turn off profile data collection. profile can be turned off by condition.

Raises

RuntimeError – If the profile has not started, this function is disabled.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, context
>>> import mindspore.dataset as ds
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics,␣
↪ExportType
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = mindspore.train.Model(net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
...     prof = mindspore.profiler.profile(activities=[ProfilerActivity.CPU,␣
↪ProfilerActivity.NPU],
...                                       schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                           repeat=1, skip_first=2),
...                                       on_trace_ready=mindspore.profiler.
...                                           tensorboard_trace_handler("./data"),
...                                       profile_memory=False,
...                                       experimental_config=experimental_config)
...     prof.start()
...     # Train Model
...     for step in range(steps):
...         train(net)
...         prof.step()
...     prof.stop()
```

mindspore.profiler._ExperimentalConfig

class mindspore.profiler.**_ExperimentalConfig**(*profiler_level: ProfilerLevel = ProfilerLevel.Level0*, *aic_metrics: AicoreMetrics = AicoreMetrics.AiCoreNone*, *l2_cache: bool = False*, *mstx: bool = False*, *data_simplification: bool = True*, *export_type: list = None*)

The purpose of this class is to configure scalable parameters when using profiles for model performance data acquisition.

Parameters

- **profiler_level** (*ProfilerLevel, optional*) –(Ascend only) The level of profiling. Default: `ProfilerLevel.Level0`.
  - ProfilerLevel.LevelNone: This setting takes effect only when mstx is enabled, indicating that no operator data is collected on the device side.

  - ProfilerLevel.Level0: Leanest level of profiling data collection, collects information about the elapsed time of the computational operators on the NPU and communication large operator information.
  - ProfilerLevel.Level1: Collect more CANN layer AscendCL data and AICore performance metrics and communication mini operator information based on Level0.
  - ProfilerLevel.Level2: Collect GE and Runtime information in CANN layer on top of Level1

- **aic_metrics** (*AicoreMetrics, optional*) –(Ascend only) Types of AICORE performance data collected, when using this parameter, *activities* must include `ProfilerActivity.NPU` , and the value must be a member of AicoreMetrics. When profiler_level is Level0, the default value is `AicoreMetrics.AiCoreNone`; Profiler_level is a Level1 or Level2 stores, the default value is: `AicoreMetrics. PipeUtilization`.The data items contained in each metric are as follows:
  - AicoreMetrics.AiCoreNone: Does not collect AICORE data.
  - AicoreMetrics.ArithmeticUtilization: ArithmeticUtilization contains mac_fp16/int8_ratio, vec_fp32/fp16/int32_ratio, vec_misc_ratio etc.
  - AicoreMetrics.PipeUtilization: PipeUtilization contains vec_ratio, mac_ratio, scalar_ratio, mte1/mte2/mte3_ratio, icache_miss_rate etc.
  - AicoreMetrics.Memory: Memory contains ub_read/write_bw, l1_read/write_bw, l2_read/write_bw, main_mem_read/write_bw etc.
  - AicoreMetrics.MemoryL0: MemoryL0 contains l0a_read/write_bw, l0b_read/write_bw, l0c_read/write_bw etc.
  - AicoreMetrics.ResourceConflictRatio: ResourceConflictRatio contains vec_bankgroup/bank/resc_cflt_ratio etc.
  - AicoreMetrics.MemoryUB: MemoryUB contains ub_read/write_bw_mte, ub_read/write_bw_vector, ub_read/write_bw_scalar etc.
  - AicoreMetrics.L2Cache: L2Cache contains write_cache_hit, write_cache_miss_allocate, r0_read_cache_hit, r1_read_cache_hit etc. This function only supports Atlas A2 training series products.
  - AicoreMetrics.MemoryAccess: Statistics on storage access bandwidth and storage capacity of main storage and l2 cache etc.
- **l2_cache** (*bool, optional*) –(Ascend only) Whether to collect l2 cache data, collect when True. Default: `False` . The l2_cache.csv file is generated in the ASCEND_PROFILER_OUTPUT folder.In O2 mode, only wait and skip_first parameters in schedule configuration can be set to 0.
- **mstx** (*bool, optional*) –(Ascend only) Whether to collect light weight profiling data, collect when True. Default: `False` .
- **data_simplification** (*bool, optional*) –(Ascend only) Whether to remove FRAMEWORK data and other redundant data. If set to True, only the profiler deliverables and raw performance data under the PROF_XXX directory are kept to save space. Default value: `True` .
- **export_type** (*list, optional*) –(Ascend only) The data type to export. The db and text formats can be exported at the same time. The default value is `None`, indicating that data of the text type is exported.
  - ExportType.Text: Export text type data.
  - ExportType.Db: Export db type data.

Raises

RuntimeError –When the version of CANN does not match the version of MindSpore, MindSpore cannot parse the generated ascend_job_id directory structure.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, context
>>> import mindspore.dataset as ds
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics, ExportType
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = mindspore.train.Model(net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
...     with mindspore.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.
↪NPU],
...                                     schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                          repeat=1, skip_first=2),
...                                     on_trace_ready=mindspore.profiler.tensorboard_trace_
↪handler("./data"),
...                                     profile_memory=False,
...                                     experimental_config=experimental_config) as prof:
...
...         # Train Model
...         for step in range(steps):
...             train(net)
...             prof.step()
```

mindspore.profiler.mstx

class mindspore.profiler.**mstx**

Mstx class provides profiling tools for marking and tracing on NPU. This class provides three static methods: mark, range_start and range_end for adding marker points and ranges in profiling.

static mark (*message: str*, *stream:* mindspore.runtime.Stream = *None*)

Add a marker point in profiling.

Parameters

- **message** (*str*) – Description for the marker.
- **stream** (*Stream*, optional) – NPU stream for async execution, expected type: mindspore.runtime.Stream. Default: None, which means only marking on host side without marking on device stream.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore
>>> from mindspore import nn
>>> import mindspore.dataset as ds
>>> from mindspore import Profiler
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, schedule,␣
↪tensorboard_trace_handler
>>> from mindspore.profiler import mstx
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield (np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32))
>>>
>>> def train(net):
...     stream = ms.runtime.current_stream()
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = ms.train.Model(net, loss, optimizer)
...     # Add marker before training
...     mstx.mark("train start", stream)
...     model.train(1, data)
...     # Add marker after training
...     mstx.mark("train end", stream)
>>>
>>> if __name__ == '__main__':
...     # Note: mstx only supports Ascend device and cannot be used in mindspore.nn.Cell.
↪construct
...     # when in mindspore.GRAPH_MODE
...     ms.set_context(mode=ms.PYNATIVE_MODE)
...     ms.set_device(device_target="Ascend", device_id=0)
```

(continues on next page)

(continued from previous page)

```
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                               profiler_level=ProfilerLevel.LevelNone,
...                               mstx=True)
...     # Note that the Profiler should be initialized before model.train
...     with mindspore.profiler.profile(
...         activities=[ProfilerActivity.CPU, ProfilerActivity.NPU],
...         schedule=schedule(wait=0, warmup=0, active=3, repeat=1, skip_first=0),
...         on_trace_ready=mindspore.profiler.tensorboard_trace_handler("./data"),
...         experimental_config=experimental_config
...     ) as profiler:
...         net = Net()
...         for i in range(5):
...             train(net)
...             profiler.step()
```

static range_end(*range_id: int*)

End a profiling range.

Parameters

range_id (*int*) –Range ID from range_start.

Examples

```
>>> # Please refer to the example in range_start
>>> # range_id = mstx.range_start("training process", stream)
>>> # model.train(1, data)
>>> # mstx.range_end(range_id)
```

static range_start(*message: str*, *stream:* mindspore.runtime.Stream = *None*)

Start a profiling range.

Parameters

- **message** (*str*) –Description for the range.
- **stream** (*Stream*, optional) –NPU stream for async execution, expected type: mindspore.runtime.Stream. Default: `None`, which means only starting mstx range on host side without starting on device stream.

Returns

int, range ID for range_end.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore
>>> from mindspore import nn
>>> import mindspore.dataset as ds
>>> from mindspore import Profiler
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, schedule,␣
↪tensorboard_trace_handler
>>> from mindspore.profiler import mstx
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield (np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32))
>>>
>>> def train(net):
...     stream = ms.runtime.current_stream()
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = ms.train.Model(net, loss, optimizer)
...     # Start profiling range
...     range_id = mstx.range_start("training process", stream)
...     model.train(1, data)
...     # End profiling range
...     mstx.range_end(range_id)
>>>
>>> if __name__ == '__main__':
...     # Note: mstx only supports Ascend device and cannot be used in mindspore.nn.Cell.
↪construct
...     # when in mindspore.GRAPH_MODE
...     ms.set_context(mode=ms.PYNATIVE_MODE)
...     ms.set_device(device_target="Ascend", device_id=0)
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.LevelNone,
...                                 mstx=True)
...     # Note that the Profiler should be initialized before model.train
...     with mindspore.profiler.profile(
...         activities=[ProfilerActivity.CPU, ProfilerActivity.NPU],
...         schedule=schedule(wait=0, warmup=0, active=3, repeat=1, skip_first=0),
...         on_trace_ready=mindspore.profiler.tensorboard_trace_handler("./data"),
...         experimental_config=experimental_config
...     ) as profiler:
...         net = Net()
...         for i in range(5):
...             train(net)
...             profiler.step()
```

mindspore.profiler.DynamicProfilerMonitor

class mindspore.profiler.**DynamicProfilerMonitor**(*cfg_path*, *output_path='./dyn_profile_data'*, *poll_interval=2*, ***kwargs*)

This class to enable the dynamic profile monitoring of MindSpore neural networks.

Parameters

- **cfg_path** (*str*) –(Ascend only) Dynamic profile json config file directory. The requirement is a shared path that can be accessed by all nodes. The parameters of the json configuration file are as follows:
  - start_step (int, required) - Sets the step number at which the Profiler starts collecting data. It is a relative value,

with the first step of training being 1. The default value is -1, indicating that data collection will not start during the entire training process.

- stop_step (int, required) - Sets the step number at which the Profiler stops collecting data. It is a relative value, with the first step of training being 1. The stop_step must be greater than or equal to start_step. The default value is -1, indicating that data collection will not start during the entire training process.
- aic_metrics (int, optional) - The range of values corresponds to the Profiler. The default value -1 indicates that AI Core utilization is not collected, and 0 indicates PipeUtilization, 1 indicates ArithmeticUtilization, 2 stands for Memory, 3 stands for MemoryL0, 4 stands for MemoryUB, 5 indicates ResourceConflictRatio, 6 indicates L2Cache, 7 indicates MemoryAccess.
- profiler_level (int, optional) - Sets the level of performance data collection, where -1 represents ProfilerLevel.LevelNone, 0 represents ProfilerLevel.Level0, 1 represents ProfilerLevel.Level1, and 2 represents ProfilerLevel.Level2. The default value is 0, indicating the ProfilerLevel.Level0 collection level.
- activities (int, optional) - Sets the devices for performance data collection, where 0 represents CPU+NPU, 1 represents CPU, and 2 represents NPU. The default value is 0, indicating the collection of CPU+NPU performance data.
- export_type (int, optional) - Sets the data type to export, where 0 represents text, 1 represents db, and 2 represents text and db. The default value is 0, indicating only export text type data.
- profile_memory (bool, optional) - Set whether to collect memory performance data, true indicates that memory performance data is collected, false indicates that memory performance data is not collected. The default value is false, indicating that memory performance data is not collected.
- mstx (bool, optional) - Set whether to enable mstx, true indicates that mstx is enabled, false indicates that mstx is disabled. The default value is false, indicating that mstx is not enabled.
- analyse_mode (int, optional) - Sets the mode for online analysis, corresponding to the analyse_mode parameter of the mindspore.Profiler.analyse interface, where 0 represents "sync" and 1 represents "async". The default value is -1, indicating that online analysis is not used.
- parallel_strategy (bool, optional) - Sets whether to collect parallel strategy performance data, where true means to collect and false means not to collect. The default value is false, indicating that parallel strategy performance data is not collected.
- with_stack (bool, optional) - Sets whether to collect call stack information, where true means to collect and false means not to collect. The default value is false, indicating that call stack information is not collected.
- data_simplification (bool, optional) - Sets whether to enable data simplification, where true means to enable and false means not to enable. The default value is true, indicating that data simplification is enabled.

- **output_path** (*str, optional*) –(Ascend only) Output data path. Default: `"./dyn_profile_data"` .
- **poll_interval** (*int, optional*) –(Ascend only) The polling period of the monitoring process, in seconds. Default value: `2`.

Raises

RuntimeError –When create shared memory times exceeds max times.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> import mindspore.dataset as ds
>>> from mindspore.profiler import DynamicProfilerMonitor
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield (np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32))
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     dynprof_cb = DynamicProfilerMonitor(cfg_path="./dyn_cfg", output_path="./dyn_prof_
↪data")
...     model = ms.train.Model(net, loss, optimizer)
...     # register DynamicProfilerMonitor to model.train()
...     model.train(10, data, callbacks=[dynprof_cb])
```

step()

Used for Ascend, distinguish step collection and parsing performance data by dynamic profiler.

Raises

RuntimeError – If the 'start_step' parameter setting is greater than the 'stop_step' parameter setting.

Examples

```
>>> import json
>>> import os
>>> import numpy as np
>>>
>>> import mindspore
>>> import mindspore.dataset as ds
>>> from mindspore import context, nn
>>> from mindspore.profiler import DynamicProfilerMonitor
>>>
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2, 2)
...
...     def construct(self, x):
...         return self.fc(x)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> def generator_net():
...     for _ in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(test_net):
...     optimizer = nn.Momentum(test_net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator_net(), ["data", "label"])
...     model = mindspore.train.Model(test_net, loss, optimizer)
...     model.train(1, data)
>>>
>>> def change_cfg_json(json_path):
...     with open(json_path, 'r', encoding='utf-8') as file:
...         data = json.load(file)
...
...     data['start_step'] = 6
...     data['stop_step'] = 7
...
...     with open(json_path, 'w', encoding='utf-8') as file:
...         json.dump(data, file, ensure_ascii=False, indent=4)
>>>
>>> if __name__ == '__main__':
...      # set json configuration file
...      context.set_context(mode=mindspore.PYNATIVE_MODE)
...      mindspore.set_device("Ascend")
...      data_cfg = {
...          "start_step": 2,
...          "stop_step": 5,
...          "aic_metrics": -1,
...          "profiler_level": 0,
...          "activities": 0,
...          "export_type": 0,
...          "profile_memory": False,
...          "mstx": False,
...          "analyse_mode": 0,
...          "parallel_strategy": False,
...          "with_stack": False,
...          "data_simplification": True,
...          }
...      output_path = "./cfg_path"
...      cfg_path = os.path.join(output_path, "profiler_config.json")
...      os.makedirs(output_path, exist_ok=True)
...      # set cfg file
...      with open(cfg_path, 'w') as f:
...           json.dump(data_cfg, f, indent=4)
...      # cfg_path contains the json configuration file path, and output_path is the␣
↪output path
...      dp = DynamicProfilerMonitor(cfg_path=output_path, output_path=output_path)
...      STEP_NUM = 15
...      # Define a network of training models
...      net = Net()
...      for i in range(STEP_NUM):
...          print(f"step {i}")
...          train(net)
...          # Modify the configuration file after step 7. For example, change start_
↪step to 8 and stop_step to 10
...          if i == 5:
```

(continues on next page)

(continued from previous page)

```
...             # Modify parameters in the JSON file
...             change_cfg_json(os.path.join(output_path, "profiler_config.json"))
...         # Call step collection
...         dp.step()
```

mindspore.profiler.schedule

class mindspore.profiler.**schedule**(*, *wait: int*, *active: int*, *warmup: int = 0*, *repeat: int = 0*, *skip_first: int = 0*)

This class use to get the actions of each step. The schedule is as follows:

```
(NONE)         (NONE)           (NONE)        (WARM_UP)        (RECORD)       (RECORD)     ␣
↪(RECORD_AND_SAVE)     None
START-------->skip_first------->wait--------->warmup-------->active........active.........
↪active----------->stop
                               |                                                           |
                               |                            repeat_1                       |
                               -------------------------------------------------------------
```

The profiler will skip the first `skip_first` steps, then wait for `wait` steps, then do the warmup for the next `warmup` steps, then do the active recording for the next `active` steps and then repeat the cycle starting with `wait` steps. The optional number of cycles is specified with the `repeat` parameter, the zero value means that the cycles will continue until the profiling is finished.

Keyword Arguments

- **wait** (`int`) – The number of steps to wait before starting the warm-up phase. must be greater than or equal to 0. If the wait parameter is not set externally, it is set to `0` when the schedule class is initialized.
- **active** (`int`) – The number of steps to record data during the active phase. must be greater than or equal to 1. If the active parameter is not set externally, it is set to `1` when the schedule class is initialized.
- **warmup** (`int, optional`) – The number of steps to perform the warm-up phase. must be greater than or equal to 0. Default value: `0`.
- **repeat** (`int, optional`) – The number of times to repeat the cycle. If repeat is set to 0, the Profiler will determine the repeat value based on the number of times the model is trained, which will generate one more performance data with incomplete collection. The data in the last step is abnormal data that users do not need to pay attention to. Default value: `0`.
- **skip_first** (`int, optional`) – The number of steps to skip at the beginning. Must be greater than or equal to 0. Default value: `0`

Raises

ValueError – When the parameter step is less than 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.dataset as ds
>>> from mindspore import context, nn
>>> from mindspore.profiler import ProfilerLevel, AicoreMetrics, ExportType, ProfilerActivity
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2, 2)
...
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator_net():
...     for _ in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(test_net):
...     optimizer = nn.Momentum(test_net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator_net(), ["data", "label"])
...     model = mindspore.train.Model(test_net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
...     with mindspore.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.
↪NPU],
...                                     schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                          repeat=1, skip_first=2),
...                                     on_trace_ready=mindspore.profiler.tensorboard_trace_
↪handler("./data"),
...                                     profile_memory=False,
...                                     experimental_config=experimental_config) as prof:
...
...         # Train Model
...         for step in range(steps):
...             train(net)
...             prof.step()
```

to_dict()
Convert schedule to a dict.

Returns

dict, the parameters of schedule and their values.

mindspore.profiler.tensorboard_trace_handler

mindspore.profiler.**tensorboard_trace_handler**(*dir_name: str = None*, *worker_name: str = None*, *analyse_flag: bool = True*, *async_mode: bool = False*)

For each step in dynamic graph mode, call this method for online analyse.

Parameters

- **dir_name** (*str, optional*) –Specifies the directory path to save the analysis results. The default is `None`. The default save path is `"./data"`.
- **worker_name** (*str, optional*) –Specifies the system version name. The default is `None`. The default project thread name is `"Name of the current operating system + process ID"`.
- **analyse_flag** (*bool, optional*) –Whether to enable online analysis. The default value is `True`. Indicates online analysis.
- **async_mode** (*bool, optional*) –Whether to use asynchronous parsing mode. The default value is `False`. Indicates the use of synchronous parsing mode.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.dataset as ds
>>> from mindspore import context, nn
>>> from mindspore.profiler import ProfilerLevel, AicoreMetrics, ExportType, ProfilerActivity
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2, 2)
...
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator_net():
...     for _ in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
>>> def train(test_net):
...     optimizer = nn.Momentum(test_net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator_net(), ["data", "label"])
...     model = mindspore.train.Model(test_net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     context.set_context(mode=mindspore.GRAPH_MODE)
```

(continues on next page)

(continued from previous page)

```
...     mindspore.set_device("Ascend")
...
...     # Init Profiler
...     experimental_config = mindspore.profiler._ExperimentalConfig(
...                                 profiler_level=ProfilerLevel.Level0,
...                                 aic_metrics=AicoreMetrics.AiCoreNone,
...                                 l2_cache=False,
...                                 mstx=False,
...                                 data_simplification=False,
...                                 export_type=[ExportType.Text])
...     steps = 10
...     net = Net()
...     # Note that the Profiler should be initialized before model.train
...     with mindspore.profiler.profile(activities=[ProfilerActivity.CPU, ProfilerActivity.
↪NPU],
...                                     schedule=mindspore.profiler.schedule(wait=1,␣
↪warmup=1, active=2,
...                                          repeat=1, skip_first=2),
...                                     on_trace_ready=mindspore.profiler.tensorboard_trace_
↪handler("./data"),
...                                     profile_memory=False,
...                                     experimental_config=experimental_config) as prof:
...
...         # Train Model
...         for step in range(steps):
...             train(net)
...             prof.step()
```

mindspore.profiler.profiler.analyse

mindspore.profiler.profiler.**analyse**(*profiler_path: str*, *max_process_number: int = os.cpu_count() // 2*, *pretty=False*, *step_list=None*, *data_simplification=True*)

Analyze training performance data offline, which is invoked after performance data collection is completed.

Parameters

- **profiler_path** (*str*) –The path to profiling data that needs to be analyzed offline, specified to the upper directory `*_ascend_ms`.
- **max_process_number** (*int, optional*) –Maximum number of processes. The default value is `os.cpu_count() // 2`.
- **pretty** (*bool, optional*) –Format the JSON file. Default: `False`, indicating that the formatting is not performed.
- **step_list** (*list, optional*) –Only the performance data of the specified step is parsed. The specified step must be a consecutive integer. It supports CallBack collection only in GRAPH mode, and can only slice the CANN layer and the following information. Default value: `None`, that is, full resolution.
- **data_simplification** (*bool, optional*) –Whether to enable data simplification. Default: `True`, indicating the data simplification is enabled.

Examples

```
>>> from mindspore.profiler.profiler import analyse
>>> analyse(profiler_path="./profiling_path")
```

mindspore.SummaryCollector

class mindspore.**SummaryCollector**(*summary_dir*, *collect_freq=10*, *num_process=32*, *collect_specified_data=None*, *keep_default_action=True*, *custom_lineage_data=None*, *collect_tensor_freq=None*, *max_file_size=None*, *export_options=None*)

SummaryCollector can help you to collect some common information, such as loss, learning late, computational graph and so on.

SummaryCollector also enables the summary operator to collect data to summary files.

Note:

1. Multiple SummaryCollector instances in callback list are not allowed.
2. Not all information is collected at the training phase or at the eval phase.
3. SummaryCollector always record the data collected by the summary operator.
4. SummaryCollector only supports Linux systems.
5. The Summary is not supported when compile source with *-s on* option.

Parameters

- **summary_dir** (*str*) –The collected data will be persisted to this directory. If the directory does not exist, it will be created automatically.
- **collect_freq** (*int*) –Set the frequency of data collection, it should be greater than zero, and the unit is *step*. If a frequency is set, we will collect data when (current steps % freq) equals to 0, and the first step will be collected at any time. It is important to note that if the data sink mode is used, the unit will become the *epoch*. It is not recommended to collect data too frequently, which can affect performance. Default: `10` .
- **num_process** (*int*) –Number of processes saving summary data. The more processes there are, the better the performance, but there may be host memory overflow issues. Default: `32` .
- **collect_specified_data** (*Union[None, dict]*) –Perform custom operations on the collected data. By default, if set to None, all data is collected as the default behavior. You can customize the collected data with a dictionary. For example, you can set {'collect_metric': False} to control not collecting metrics. The data that supports control is shown below. Default: `None` .
  - collect_metric (bool): Whether to collect training metrics, currently only the loss is collected. The first output will be treated as the loss and it will be averaged. Default: `True` .
  - collect_graph (bool): Whether to collect the computational graph. Currently, only training computational graph is collected. Default: `True` .
  - collect_train_lineage (bool): Whether to collect lineage data for the training phase. Default: `True` .
  - collect_eval_lineage (bool): Whether to collect lineage data for the evaluation phase. Default: `True` .
  - collect_input_data (bool): Whether to collect dataset for each training. Currently only image data is supported. If there are multiple columns of data in the dataset, the first column should be image data. Default: `True` .
  - collect_dataset_graph (bool): Whether to collect dataset graph for the training phase. Default: `True` .

- histogram_regular (Union[str, None]): Collect weight and bias for parameter distribution page. This field allows regular strings to control which parameters to collect. It is not recommended to collect too many parameters at once, as it can affect performance. Note that if you collect too many parameters and run out of memory, the training will fail. Default: `None` , it means only the first five parameters are collected.
- collect_landscape (Union[dict,None]): Whether to collect the parameters needed to create the loss landscape. If set to None, collect_landscape parameters will not be collected. All parameter information is collected by default and stored in file *{summary_dir}/ckpt_dir/train_metadata.json*.
  - landscape_size (int): Specify the image resolution of the generated loss landscape. For example, if it is set to `128` , the resolution of the landscape is 128 * 128. The calculation time increases with the increase of resolution. Default: `40` . Optional values: between 3 and 256.
  - unit (str): Specify the interval strength of the training process. Default: `"step"` . Optional: epoch/step.
  - create_landscape (dict): Select how to create loss landscape. Training process loss landscape(train) and training result loss landscape(result). Default: `{"train": True, "result": True}`. Optional: `True` / `False` .
  - num_samples (int): The size of the dataset used to create the loss landscape. For example, in image dataset, You can set num_samples is 128, which means that 128 images are used to create loss landscape. Default: `128` .
  - intervals (List[List[int]]): Specifies the interval in which the loss landscape. For example: If the user wants to create loss landscape of two training processes, they are 1-5 epoch and 6-10 epoch respectively. They anc set [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]. Note: Each interval have at least three epochs.

- **keep_default_action** (*bool*) – This field affects the collection behavior of the 'collect_specified_data' field. True: it means that after specified data is set, non-specified data is collected as the default behavior. False: it means that after specified data is set, only the specified data is collected, and the others are not collected. Default: `True` .
- **custom_lineage_data** (*Union[dict, None]*) – Allows you to customize the data. In the custom data, the type of the key supports str, and the type of value supports str, int and float. Default: `None` , it means there is no custom data.
- **collect_tensor_freq** (*Optional[int]*) – The same semantics as the *collect_freq*, but controls TensorSummary only. Because TensorSummary data is too large to be compared with other summary data, this parameter is used to reduce its collection. By default, The maximum number of steps for collecting TensorSummary data is 20, but it will not exceed the number of steps for collecting other summary data. For example, given *collect_freq=10*, when the total steps is 600, TensorSummary will be collected 20 steps, while other summary data 61 steps, but when the total steps is 20, both TensorSummary and other summary will be collected 3 steps. Also note that when in parallel mode, the total steps will be split evenly, which will affect the number of steps TensorSummary will be collected. Default: `None` , which means to follow the behavior as described above.
- **max_file_size** (*Optional[int]*) – The maximum size in bytes of each file that can be written to the disk. For example, to write not larger than 4GB, specify *max_file_size=4*1024*3*. Default: `None` , which means no limit.
- **export_options** (*Union[None, dict]*) – Perform custom operations on the export data. Note that the size of export files is not limited by the max_file_size. You can customize the export data with a dictionary. For example, you can set {'tensor_format': 'npy'} to export tensor as npy file. The data that supports control is shown below. Default: `None` , it means that the data is not exported.
  - tensor_format (Union[str, None]): Customize the export tensor format. Supports ["npy", None]. Default: `None` , it means that the tensor is not exported.
    - npy: export tensor as npy file.

Raises

ValueError – The Summary is not supported, please without *-s on* and recompile source.

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn, SummaryCollector
>>> from mindspore.train import Model, Accuracy
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     ms.set_context(mode=ms.GRAPH_MODE, device_target="Ascend")
...     mnist_dataset_dir = '/path/to/mnist_dataset_directory'
...     # Create the dataset taking MNIST as an example. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
...     ds_train = create_dataset()
...     # Define the network structure of LeNet5. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
...     network = LeNet5(10)
...     net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
...     net_opt = nn.Momentum(network.trainable_params(), 0.01, 0.9)
...     model = Model(network, net_loss, net_opt, metrics={"Accuracy": Accuracy()}, amp_
↪level="O2")
...
...     # Simple usage:
...     summary_collector = SummaryCollector(summary_dir='./summary_dir')
...     model.train(1, ds_train, callbacks=[summary_collector], dataset_sink_mode=False)
...
...     # Do not collect metric and collect the first layer parameter, others are collected␣
↪by default
...     specified={'collect_metric': False, 'histogram_regular': '^conv1.*'}
...     summary_collector = SummaryCollector(summary_dir='./summary_dir', collect_specified_
↪data=specified)
...     model.train(1, ds_train, callbacks=[summary_collector], dataset_sink_mode=False)
```

mindspore.SummaryLandscape

class mindspore.**SummaryLandscape**(*summary_dir*)

SummaryLandscape can help you to collect loss landscape information. It can create landscape in PCA direction or random direction by calculating loss.

Note: SummaryLandscape only supports Linux systems.

Parameters

summary_dir (*str*) – The path of summary is used to save the model weight, metadata and other data required to create landscape.

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore.train import Model, Accuracy, Loss
>>> from mindspore import SummaryCollector, SummaryLandscape
>>>
>>> if __name__ == '__main__':
...     # If the device_target is Ascend, set the device_target to "Ascend"
...     ms.set_context(mode=ms.GRAPH_MODE, device_target="GPU")
...     # Create the dataset taking MNIST as an example. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
...     ds_train = create_dataset()
...     # Define the network structure of LeNet5. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
...     network = LeNet5()
...     net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
...     net_opt = nn.Momentum(network.trainable_params(), 0.01, 0.9)
...     model = Model(network, net_loss, net_opt, metrics={"Accuracy": Accuracy()})
...     # Simple usage for collect landscape information:
...     interval_1 = [1, 2, 3, 4, 5]
...     summary_collector = SummaryCollector(summary_dir='./summary/lenet_interval_1',
...                                          collect_specified_data={'collect_landscape':{
↪"landscape_size": 4,
...
↪"unit": "step",
...                                                                  "create_
↪landscape":{"train":True,
...                                                                           ␣
↪   "result":False},
...                                                                  "num_samples":␣
↪2048,
...                                                                  "intervals":␣
↪[interval_1]}
...                                                                 })
...     model.train(1, ds_train, callbacks=[summary_collector], dataset_sink_mode=False)
...
...     # Simple usage for visualization landscape:
...     def callback_fn():
...         # Define the network structure of LeNet5. Refer to
...         # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
...         network = LeNet5()
...         net_loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction="mean")
...         metrics = {"Loss": Loss()}
...         model = Model(network, net_loss, metrics=metrics)
...         # Create the dataset taking MNIST as an example. Refer to
...         # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
...         ds_eval = create_dataset()
...         return model, network, ds_eval, metrics
...
...     summary_landscape = SummaryLandscape('./summary/lenet_interval_1')
...     # parameters of collect_landscape can be modified or unchanged
...     summary_landscape.gen_landscapes_with_multi_process(callback_fn,
...                                                         collect_landscape={"landscape_size
↪": 4,
...                                                                            "create_landscape
↪":{"train":False,
```

(continues on next page)

(continued from previous page)

```
...                                                                                                 ␣
↪  "result":False},
...                                                                                  "num_samples":␣
↪2048,
...                                                                                  "intervals":␣
↪[interval_1]},
...                                                                 device_ids=[1])
```

clean_ckpt()
Clean the checkpoint.

gen_landscapes_with_multi_process(*callback_fn*, *collect_landscape=None*, *device_ids=None*, *output=None*)
Use the multi process to generate landscape.

Parameters

- **callback_fn** (*python function*) –A python function object. User needs to write a function, it has no input, and the return requirements are as follows.
  - mindspore.train.Model: User's model object.
  - mindspore.nn.Cell: User's network object.
  - mindspore.dataset: User's dataset object for create loss landscape.
  - mindspore.train.Metrics: User's metrics object.
- **collect_landscape** (*Union[dict, None]*) –The meaning of the parameters when creating loss landscape is consistent with the fields with the same name in SummaryCollector. The purpose of setting here is to allow users to freely modify creating parameters. Default: `None` .
  - landscape_size (int): Specify the image resolution of the generated loss landscape. For example, if it is set to `128` , the resolution of the landscape is 128 * 128. The calculation time increases with the increase of resolution. Default: `40` . Optional values: between 3 and 256.
  - create_landscape (dict): Select how to create loss landscape. Training process loss landscape(train) and training result loss landscape(result). Default: `{"train": True, "result": True}`. Optional: `True` / `False` .
  - num_samples (int): The size of the dataset used to create the loss landscape. For example, in image dataset, You can set num_samples is 2048, which means that 2048 images are used to create loss landscape. Default: `2048` .
  - intervals (List[List[int]]): Specifies the interval in which the loss landscape. For example: If the user wants to create loss landscape of two training processes, they are 1-5 epoch and 6-10 epoch respectively. They can set [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]. Note: Each interval have at least three epochs.
- **device_ids** (*List(int)*) –Specifies which devices are used to create loss landscape. For example: [0, 1] refers to creating loss landscape with device 0 and device 1. Default: `None` .
- **output** (*str*) –Specifies the path to save the loss landscape. Default: `None` . The default save path is the same as the summary file.

mindspore.SummaryRecord

class mindspore.**SummaryRecord**(*log_dir*, *file_prefix='events'*, *file_suffix='_MS'*, *num_process=32*, *network=None*, *max_file_size=None*, *raise_exception=False*, *export_options=None*)

SummaryRecord is used to record the summary data and lineage data.

The API will create a summary file and lineage files lazily in a given directory and writes data to them. It writes the data to files by executing the *record* method. In addition to recording the data bubbled up from the network by defining the summary operators, SummaryRecord also supports to record extra data which can be added by calling add_value.

Note:

1. Make sure to close the SummaryRecord at the end, otherwise the process will not exit. Please see the Example section below to learn how to close properly in two ways.
2. Only one SummaryRecord instance is allowed at a time, otherwise it will cause data writing problems.
3. SummaryRecord only supports Linux systems.
4. The Summary is not supported when compile source with *-s on* option.

Parameters

- **log_dir** (*str*) –The log_dir is a directory location to save the summary.
- **file_prefix** (*str*) –The prefix of file. Default: `"events"` .
- **file_suffix** (*str*) –The suffix of file. Default: `"_MS"` .
- **num_process** (*int*) –Number of processes saving summary data. The more processes there are, the better the performance, but there may be host memory overflow issues. Default: `32` .
- **network** (`Cell`) –Obtain a pipeline through network for saving graph summary. Default: `None` .
- **max_file_size** (*int, optional*) –The maximum size of each file that can be written to disk (in bytes). For example, to write not larger than 4GB, specify *max_file_size=4*1024**3*. Default: `None` , which means no limit.
- **raise_exception** (*bool, optional*) –Sets whether to throw an exception when a RuntimeError or OSError exception occurs in recording data. Default: `False` , this means that error logs are printed and no exception is thrown.
- **export_options** (*Union[None, dict]*) –Perform custom operations on the export data. Note that the size of export files is not limited by the max_file_size. You can customize the export data with a dictionary. For example, you can set {'tensor_format': 'npy'} to export tensor as npy file. The data that supports control is shown below. Default: `None` , it means that the data is not exported.
  - tensor_format (Union[str, None]): Customize the export tensor format. Supports ["npy", None]. Default: `None` , it means that the tensor is not exported.
    - npy: export tensor as npy file.

Raises

- **TypeError** –*max_file_size* is not int or *file_prefix* and *file_suffix* is not string.
- **ValueError** –The Summary is not supported, please without *-s on* and recompile source.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     # use in with statement to auto close
...     with ms.SummaryRecord(log_dir="./summary_dir") as summary_record:
...         pass
...
...     # use in try .. finally .. to ensure closing
...     try:
...         summary_record = ms.SummaryRecord(log_dir="./summary_dir")
...     finally:
...         summary_record.close()
```

add_value (*plugin*, *name*, *value*)

Add value to be recorded later.

Parameters

- **plugin** (*str*) – The plugin of the value.
  - graph: the value is a computational graph.
  - scalar: the value is a scalar.
  - image: the value is an image.
  - tensor: the value is a tensor.
  - histogram: the value is a histogram.
  - train_lineage: the value is a lineage data for the training phase.
  - eval_lineage: the value is a lineage data for the evaluation phase.
  - dataset_graph: the value is a dataset graph.
  - custom_lineage_data: the value is a customized lineage data.
  - LANDSCAPE: the value is a landscape.
- **name** (*str*) – The value of the name.
- **value** (*Union[Tensor, GraphProto, TrainLineage, EvaluationLineage, DatasetGraph, UserDefinedInfo, LossLandscape]*) – The value to store.
  - The data type of value should be 'GraphProto' (see mindspore/ccsrc/anf_ir.proto) object when the plugin is 'graph'.
  - The data type of value should be 'Tensor' object when the plugin is 'scalar', 'image', 'tensor' or 'histogram'.
  - The data type of value should be a 'TrainLineage' object when the plugin is 'train_lineage', see mindspore/ccsrc/lineage.proto.
  - The data type of value should be a 'EvaluationLineage' object when the plugin is 'eval_lineage', see mindspore/ccsrc/lineage.proto.
  - The data type of value should be a 'DatasetGraph' object when the plugin is 'dataset_graph', see mindspore/ccsrc/lineage.proto.
  - The data type of value should be a 'UserDefinedInfo' object when the plugin is 'custom_lineage_data', see mindspore/ccsrc/lineage.proto.
  - The data type of value should be a 'LossLandscape' object when the plugin is 'LANDSCAPE', see mindspore/ccsrc/summary.proto.

Raises

- **ValueError** – *plugin* is not in the optional value.
- **TypeError** – *name* is not non-empty string, or the data type of value is not 'Tensor' object when the plugin is 'scalar', 'image', 'tensor' or 'histogram'.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> if __name__ == '__main__':
...     with ms.SummaryRecord(log_dir="./summary_dir", file_prefix="xx_", file_suffix="_
↪yy") \
...             as summary_record:
...         summary_record.add_value('scalar', 'loss', Tensor(0.1))
```

close()

Flush the buffer and write files to disk and close summary records. Please use the statement to autoclose.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     try:
...         summary_record = ms.SummaryRecord(log_dir="./summary_dir")
...     finally:
...         summary_record.close()
```

flush()

Flush the buffer and write buffer data to disk.

Call it to make sure that all pending events have been written to disk.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     with ms.SummaryRecord(log_dir="./summary_dir", file_prefix="xx_", file_suffix="_
↪yy") \
...             as summary_record:
...         summary_record.flush()
```

property log_dir

Get the full path of the log file.

Returns

str, the full path of log file.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     with ms.SummaryRecord(log_dir="./summary_dir", file_prefix="xx_", file_suffix="_
↪yy") \
...             as summary_record:
...         log_dir = summary_record.log_dir
```

record(*step*, *train_network=None*, *plugin_filter=None*)

Record the summary.

Parameters

- **step** (*int*) –Represents training step number.
- **train_network** (Cell) –The spare network for saving graph. Default: None, it means just do not save the graph summary when the original network graph is None.
- **plugin_filter** (*Callable[[str], bool], optional*) –The filter function, which is used to filter out which plugin should be written. Default: None.

Returns

bool, whether the record process is successful or not.

Raises

TypeError –*step* is not int, or *train_network* is not *mindspore.nn.Cell*.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     with ms.SummaryRecord(log_dir="./summary_dir", file_prefix="xx_", file_suffix="_
↪yy") \
...             as summary_record:
...         result = summary_record.record(step=2)
```

set_mode(*mode*)

Set the model running phase. Different phases affect data recording.

Parameters

mode (*str*) –The mode to be set, which should be 'train' or 'eval'. When the mode is 'eval', summary_record will not record the data of summary operators.

- train: the model running phase is train mode.
- eval: the model running phase is eval mode, When the mode is 'eval', summary_record will not record the data of summary operators.

Raises

ValueError –*mode* is not in the optional value.

Examples

```
>>> import mindspore as ms
>>> if __name__ == '__main__':
...     with ms.SummaryRecord(log_dir="./summary_dir", file_prefix="xx_", file_suffix="_
↪yy") \
...             as summary_record:
...         summary_record.set_mode('eval')
```

mindspore.set_dump

mindspore.**set_dump**(*target*, *enabled=True*)

Enable or disable dump for the *target* and its contents.

target should be an instance of *mindspore.nn.Cell* or *mindspore.ops.Primitive* . Please note that this API takes effect only when Synchronous Dump is enabled and the *dump_mode* field in dump config file is "2" . See the dump document for details. The default enabled status for a *mindspore.nn.Cell* or *mindspore.ops.Primitive* is False.

Note:

1. This API is only effective for GRAPH_MODE whose graph compilation level is O0/O1 with Ascend backend, and can not work for fusion Primitive operators.
2. This API only supports being called before training starts. If you call this API during training, it may not be effective.
3. After using *set_dump(Cell, True)* , operators in forward and backward computation (computation generated by the grad operations) of the cell will be dumped.
4. For *mindspore.nn.SoftmaxCrossEntropyWithLogits* layer, the forward computation and backward computation use the same set of operators. So you can only see dump data from backward computation. Please note that *mindspore.nn.SoftmaxCrossEntropyWithLogits* layer will also use the above operators internally when initialized with *sparse=True* and *reduction="mean"* .

Parameters

- **target** (*Union[Cell, Primitive]*) – The Cell instance or Primitive instance to which the dump flag is set.
- **enabled** (*bool, optional*) – True means enable dump, False means disable dump. Default: True .

Supported Platforms:

Ascend

Examples

Note: Please set environment variable *MINDSPORE_DUMP_CONFIG* to the dump config file and set *dump_mode* field in dump config file to 2 before running this example. See dump document for details.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor, set_dump
```

(continues on next page)

(continued from previous page)

```
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>>
>>> class MyNet(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.conv1 = nn.Conv2d(5, 6, 5, pad_mode='valid')
...         self.relu1 = nn.ReLU()
...
...     def construct(self, x):
...         x = self.conv1(x)
...         x = self.relu1(x)
...         return x
>>>
>>> if __name__ == "__main__":
...     net = MyNet()
...     set_dump(net.conv1)
...     input_tensor = Tensor(np.ones([1, 5, 10, 10], dtype=np.float32))
...     output = net(input_tensor)
```

mindspore.Profiler

class mindspore.**Profiler**(***kwargs*)

The current interface is deprecated, please use: *mindspore.profiler.profile* instead. This class to enable the profiling of MindSpore neural networks. MindSpore users can import the mindspore.Profiler, initialize the Profiler object to start profiling, and use Profiler.analyse() to stop profiling and analyse the results. Users can visualize the results using the MindStudio Insight tool. Now, Profiler supports AICORE operator, AICPU operator, HostCPU operator, memory, correspondence, cluster, etc data analysis.

Parameters

- **start_profile** (*bool, optional*) –The start_profile parameter controls whether to enable or disable performance data collection based on conditions. Default: `True` .
- **output_path** (*str, optional*) –Output data path. Default: `"./data"` .
- **profiler_level** (*ProfilerLevel, optional*) –(Ascend only) The level of profiling. Default: `ProfilerLevel.Level0`.
  - ProfilerLevel.LevelNone: This setting takes effect only when mstx is enabled, indicating that no operator data is collected on the device side.
  - ProfilerLevel.Level0: Leanest level of profiling data collection, collects information about the elapsed time of the computational operators on the NPU and communication large operator information.
  - ProfilerLevel.Level1: Collect more CANN layer AscendCL data and AICore performance metrics and communication mini operator information based on Level0.
  - ProfilerLevel.Level2: Collect GE and Runtime information in CANN layer on top of Level1
- **activities** (*list, optional*) –The activities to collect. Default: `[ProfilerActivity.CPU, ProfilerActivity.NPU]`.
  - ProfilerActivity.CPU: Collect MindSpore framework data.
  - ProfilerActivity.NPU: Collect CANN software stack and NPU data.
  - ProfilerActivity.GPU: Collect GPU data.

- **schedule** (schedule, *optional*) –Sets the action strategy for the capture, defined by the schedule class, to be used with the step interface. Default: None. Performance data of all steps is collected. For details, see *mindspore.profiler.schedule* .
- **on_trace_ready** (*Callable, optional*) –Sets the callback function to be executed when the performance data is collected. Default: None. It indicates that only performance data is collected, but not resolved. For details, see *mindspore.profiler.tensorboard _trace_handler()* .
- **profile_memory** (*bool, optional*) –(Ascend only) Whether to collect tensor memory data, collect when True . When using this parameter, *activities* must set to [ProfilerActivity.CPU, ProfilerActivity.NPU]. Collecting operator memory data when the graph compilation level is O2 requires collecting from the first step. Default: False . The operator name currently collected by this parameter is incomplete. This issue will be resolved in later versions. It is recommended to use the environment variable MS_ALLOC_CONF instead.
- **aic_metrics** (*AicoreMetrics, optional*) –(Ascend only) Types of AICORE performance data collected, when using this parameter, *activities* must include ProfilerActivity.NPU , and the value must be a member of AicoreMetrics. When *profiler_level* is ProfilerLevel.Level0, the default value is AicoreMetrics.AiCoreNone; when *profiler_level* is ProfilerLevel.Level1 or ProfilerLevel.Level2, the default value is AicoreMetrics.PipeUtilization.

  The data items contained in each metric are as follows:

  - AicoreMetrics.AiCoreNone: Does not collect AICORE data.
  - AicoreMetrics.ArithmeticUtilization: ArithmeticUtilization contains mac_fp16/int8_ratio, vec_fp32/fp16/int32_ratio, vec_misc_ratio etc.
  - AicoreMetrics.PipeUtilization: PipeUtilization contains vec_ratio, mac_ratio, scalar_ratio, mte1/mte2/mte3_ratio, icache_miss_rate etc.
  - AicoreMetrics.Memory: Memory contains ub_read/write_bw, l1_read/write_bw, l2_read/write_bw, main_mem_read/write_bw etc.
  - AicoreMetrics.MemoryL0: MemoryL0 contains l0a_read/write_bw, l0b_read/write_bw, l0c_read/write_bw etc.
  - AicoreMetrics.ResourceConflictRatio: ResourceConflictRatio contains vec_bankgroup/bank/resc_cflt_ratio etc.
  - AicoreMetrics.MemoryUB: MemoryUB contains ub_read/write_bw_mte, ub_read/write_bw_vector, ub_/write_bw_scalar etc.
  - AicoreMetrics.L2Cache: L2Cache contains write_cache_hit, write_cache_miss_allocate, r0_read_cache_hit, r1_read_cache_hit etc. This function only support Atlas A2 training series products.
  - AicoreMetrics.MemoryAccess: Statistics on storage access bandwidth and storage capacity of main storage and l2 cache etc.
- **with_stack** (*bool, optional*) –(Ascend only) Whether to collect frame host call stack data on the Python side. This data is presented in the form of a flame graph in the timeline. When using this parameter, *activities* must include ProfilerActivity.CPU. Default value: False .
- **data_simplification** (*bool, optional*) –(Ascend only) Whether to remove FRAMEWORK data and other redundant data. If set to True, only the profiler deliverables and raw performance data under the PROF_XXX directory are kept to save space. Default value: True .
- **l2_cache** (*bool, optional*) –(Ascend only) Whether to collect l2 cache data, collect when True. Default: False . The l2_cache.csv file is generated in the ASCEND_PROFILER_OUTPUT folder.In O2 mode, only wait and skip_first parameters in schedule configuration can be set to 0.
- **hbm_ddr** (*bool, optional*) –(Ascend only) Whether to collect On-Chip Memory/DDR read and write rate data, collect when True. Default: False .
- **pcie** (*bool, optional*) –(Ascend only) Whether to collect PCIe bandwidth data, collect when True. Default: False .

- **data_process** (*bool, optional*) –(Ascend/GPU) Whether to collect data to prepare performance data. Default value: `False` .
- **parallel_strategy** (*bool, optional*) –(Ascend only) Whether to collect parallel policy performance data. Default value: `False` .
- **sync_enable** (*bool, optional*) –(GPU only) Whether the profiler collects operators in a synchronous way. Default: `True` .
  - True: The synchronous way. Before sending the operator to the GPU, the CPU records the start timestamp. Then the operator is returned to the CPU after execution, and the end timestamp is recorded, The duration of the operator is the difference between the two timestamps.
  - False: The asynchronous way. The duration of the operator is that of sending from the CPU to the GPU. This method can reduce the impact of adding profiler on overall training time.

Raises

RuntimeError –When the version of CANN does not match the version of MindSpore, MindSpore cannot parse the generated ascend_job_id directory structure.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn
>>> import mindspore.dataset as ds
>>> from mindspore import Profiler
>>> from mindspore.profiler import ProfilerLevel, ProfilerActivity, AicoreMetrics
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2,2)
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator():
...     for i in range(2):
...         yield (np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32))
>>>
>>> def train(net):
...     optimizer = nn.Momentum(net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator, ["data", "label"])
...     model = ms.train.Model(net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     # If the device_target is GPU, set the device_target to "GPU"
...     ms.set_context(mode=ms.GRAPH_MODE, device_target="Ascend")
...
...     # Init Profiler
...     # Note that the Profiler should be initialized before model.train
```

(continues on next page)

(continued from previous page)

```
...     profiler = Profiler(profiler_level=ProfilerLevel.Level0,
...                         activities=[ProfilerActivity.CPU, ProfilerActivity.NPU],
...                         aic_metrics=AicoreMetrics.AiCoreNone)
...
...     # Train Model
...     net = Net()
...     train(net)
...
...     # Profiler end
...     profiler.analyse()
```

add_metadata(*key: str*, *value: str*)

Report custom metadata key-value pair data.

Parameters

- **key** (*str*) – The key to the metadata.
- **value** (*str*) – The value to the metadata.

Examples

```
>>> from mindspore import Profiler
>>> # Profiler init.
>>> profiler = Profiler()
>>> # Call Profiler add_metadata
>>> profiler.add_metadata("test_key", "test_value")
>>> # Profiler end
>>> profiler.stop()
```

add_metadata_json(*key: str*, *value: str*)

Report custom metadata key-value pair data with the value as a JSON string data.

Parameters

- **key** (*str*) – The key to the metadata.
- **value** (*str*) – The json str format value to the metadata.

Examples

```
>>> import json
>>> from mindspore import Profiler
>>> # Profiler init.
>>> profiler = Profiler()
>>> # Call Profiler add_metadata_json
>>> profiler.add_metadata_json("test_key", json.dumps({"key1": 1, "key2": 2}))
>>> # Profiler end, metadata will be saved in profiler_metadata.json
>>> profiler.stop()
```

analyse(*offline_path=None*, *pretty=False*, *step_list=None*, *mode='sync'*)

Collect and analyze training performance data, support calls during and after training. The example shows above.

Parameters

- **offline_path** (*Union[str, None], optional*) –The data path which need to be analyzed with offline mode. Offline mode isused in abnormal exit scenario. This parameter should be set to `None` for online mode. Default: `None`.
- **pretty** (*bool, optional*) –Whether to pretty json files. Default: `False`.
- **step_list** (*list, optional*) –A list of steps that need to be analyzed, the steps must be consecutive integers. Default: `None`. By default, all steps will be analyzed.
- **mode** (*str, optional*) –Analysis mode, it must be one of ["sync", "async"]. Default: `sync`.
  - sync: analyse data in current process, it will block the current process.
  - async: analyse data in subprocess, it will not block the current process. Since the parsing process will take up extra CPU resources, please enable this mode according to the actual resource situation.

Examples

```
>>> from mindspore.train import Callback
>>> from mindspore import Profiler
>>> class StopAtStep(Callback):
...     def __init__(self, start_step=1, stop_step=5):
...         super(StopAtStep, self).__init__()
...         self.start_step = start_step
...         self.stop_step = stop_step
...         self.profiler = Profiler(start_profile=False)
...
...     def step_begin(self, run_context):
...         cb_params = run_context.original_args()
...         step_num = cb_params.cur_step_num
...         if step_num == self.start_step:
...             self.profiler.start()
...
...     def step_end(self, run_context):
...         cb_params = run_context.original_args()
...         step_num = cb_params.cur_step_num
...         if step_num == self.stop_step:
...             self.profiler.stop()
...
...     def end(self, run_context):
...         self.profiler.analyse(step_list=[2,3,4], mode="sync")
```

classmethod offline_analyse(*path: str*, *pretty=False*, *step_list=None*, *data_simplification=True*)

Analyze training performance data offline, which is invoked after performance data collection is completed.

Parameters

- **path** (*str*) –The profiling data path which needs to be analyzed offline. There needs to be a profiler directory in this path.
- **pretty** (*bool, optional*) –Whether to pretty json files. Default: `False`.
- **step_list** (*list, optional*) –A list of steps that need to be analyzed, the steps must be consecutive integers. Default: `None`. By default, all steps will be analyzed.
- **data_simplification** (*bool, optional*) –Whether to enable data simplification. Default: `True`.

Examples

```
>>> from mindspore import Profiler
>>> Profiler.offline_analyse("./profiling_path")
```

op_analyse(*op_name*, *device_id=None*)

Profiler users can use this interface to obtain operator performance data.

Parameters

- **op_name** (*str or list*) –The primitive operator name to query.
- **device_id** (*int, optional*) –ID of the target device. This parameter is optional during network training or inference, and users can use device_id parameter to specify which card operator performance data to parse. If this interface is used for offline data parsing, the default value is `None` .

Raises

- **TypeError** –If the *op_name* parameter type is incorrect.
- **TypeError** –If the *device_id* parameter type is incorrect.
- **RuntimeError** –If MindSpore runs on Ascend, this interface cannot be used.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Profiler
>>> from mindspore import nn
>>> from mindspore import Model
>>> # Profiler init.
>>> profiler = Profiler()
>>> # Train Model or eval Model, taking LeNet5 as an example.
>>> # Refer to https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> # Create the dataset taking MNIST as an example.
>>> # Refer to https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataloader = create_dataset()
>>> model = Model(net, loss, optimizer)
>>> model.train(5, dataloader, dataset_sink_mode=False)
>>>
>>> # Profiler end
>>> profiler.analyse()
>>>
>>> profiler.op_analyse(op_name=["BiasAdd", "Conv2D"])
```

start()

Turn on Profiler data collection. Profiler can be turned on by condition.

Raises

- **RuntimeError** –If the profiler has already started.
- **RuntimeError** –If the *start_profile* parameter is not set or is set to `True`.

Examples

```
>>> from mindspore.train import Callback
>>> from mindspore import Profiler
>>> class StopAtStep(Callback):
...     def __init__(self, start_step, stop_step):
...         super(StopAtStep, self).__init__()
...         self.start_step = start_step
...         self.stop_step = stop_step
...         self.profiler = Profiler(start_profile=False)
...
...     def step_begin(self, run_context):
...         cb_params = run_context.original_args()
...         step_num = cb_params.cur_step_num
...         if step_num == self.start_step:
...             self.profiler.start()
...
...     def step_end(self, run_context):
...         cb_params = run_context.original_args()
...         step_num = cb_params.cur_step_num
...         if step_num == self.stop_step:
...             self.profiler.stop()
...
...     def end(self, run_context):
...         self.profiler.analyse()
```

step()

Used for Ascend, distinguish step collection and parsing performance data through schedule and on_trace_ready.

Raises

- **RuntimeError** – If the *start_profile* parameter is not set or the Profiler is not started.
- **RuntimeError** – If the *schedule* parameter is not set.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import context, nn, Profiler
>>> from mindspore.profiler import schedule, tensorboard_trace_handler, ProfilerLevel,␣
↪AicoreMetrics,
>>> ExportType, ProfilerActivity
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(2, 2)
...
...     def construct(self, x):
...         return self.fc(x)
>>>
>>> def generator_net():
...     for _ in range(2):
...         yield np.ones([2, 2]).astype(np.float32), np.ones([2]).astype(np.int32)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> def train(test_net):
...     optimizer = nn.Momentum(test_net.trainable_params(), 1, 0.9)
...     loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
...     data = ds.GeneratorDataset(generator_net(), ["data", "label"])
...     model = ms.train.Model(test_net, loss, optimizer)
...     model.train(1, data)
>>>
>>> if __name__ == '__main__':
...     context.set_context(mode=ms.PYNATIVE_MODE, device_target="Ascend")
...
...     net = Net()
...     STEP_NUM = 15
...
...     with Profiler(schedule=schedule(wait=1, warmup=1, active=2, repeat=1, skip_
↪first=2),
...                   on_trace_ready=tensorboard_trace_handler) as prof:
...         for _ in range(STEP_NUM):
...             train(net)
...             prof.step()
```

stop()

Turn off Profiler data collection. Profiler can be turned off by condition.

Raises

RuntimeError – If the profiler has not started, this function is disabled.

Examples

```
>>> from mindspore.train import Callback
>>> from mindspore import Profiler
>>> class StopAtEpoch(Callback):
...     def __init__(self, start_epoch, stop_epoch):
...         super(StopAtEpoch, self).__init__()
...         self.start_epoch = start_epoch
...         self.stop_epoch = stop_epoch
...         self.profiler = Profiler(start_profile=False)
...
...     def epoch_begin(self, run_context):
...         cb_params = run_context.original_args()
...         epoch_num = cb_params.cur_epoch_num
...         if epoch_num == self.start_epoch:
...             self.profiler.start()
...
...     def epoch_end(self, run_context):
...         cb_params = run_context.original_args()
...         epoch_num = cb_params.cur_epoch_num
...         if epoch_num == self.stop_epoch:
...             self.profiler.stop()
...
...     def end(self, run_context):
...         self.profiler.analyse()
```

1.9.3 Log

mindspore.get_level	Get the logger level.
mindspore.get_log_config	Get logger configurations.

mindspore.get_level

mindspore.**get_level**()

Get the logger level.

Returns

str, the Log level includes 4(CRITICAL), 3(ERROR), 2(WARNING), 1(INFO), 0(DEBUG).

Examples

```
>>> import os
>>> os.environ['GLOG_v'] = '3'
>>> import mindspore as ms
>>> level = ms.get_level()
>>> print(level)
'3'
```

mindspore.get_log_config

mindspore.**get_log_config**()

Get logger configurations.

Returns

Dict, the dictionary of logger configurations.

Examples

```
>>> import os
>>> import mindspore as ms
>>> os.environ['GLOG_v'] = '1'
>>> os.environ['GLOG_logtostderr'] = '0'
>>> os.environ['GLOG_log_dir'] = '/var/log'
>>> os.environ['logger_maxBytes'] = '5242880'
>>> os.environ['logger_backupCount'] = '10'
>>> os.environ['GLOG_stderrthreshold'] = '2'
>>> config = ms.get_log_config()
>>> print(config)
{'GLOG_v': '1', 'GLOG_logtostderr': '0', 'GLOG_log_dir': '/var/log',
 'logger_maxBytes': '5242880', 'logger_backupCount': '10', 'GLOG_stderrthreshold': '2'}
```

1.9.4 Installation Verification

mindspore.run_check	Provide a convenient API to check if the installation is successful or failed.

mindspore.run_check

mindspore.**run_check**()

Provide a convenient API to check if the installation is successful or failed. If the version in the check result is not as expected, use *mindspore.set_context()* to set device_target before run_check().

Examples

```
>>> import mindspore
>>> mindspore.run_check()
MindSpore version: xxx
The result of multiplication calculation is correct, MindSpore has been installed on␣
↪platform [XX] successfully!
```

1.9.5 Security

mindspore.obfuscate_ckpt	Obfuscate the plaintext checkpoint files according to the obfuscation config.
mindspore.load _obf_params_into_net	Modify model structure according to obfuscation config and load obfuscated checkpoint into obfuscated network.

mindspore.obfuscate_ckpt

mindspore.**obfuscate_ckpt**(*network*, *ckpt_files*, *target_modules=None*, *obf_config=None*, *saved_path='./'*, *obfuscate_scale=100*)

Obfuscate the plaintext checkpoint files according to the obfuscation config.

Parameters

- **network** (nn.Cell) –The original network that need to be obfuscated.
- **ckpt_files** (*str*) –The directory path of original ckpt files.
- **target_modules** (*list[str], optional*) –The target ops that need to be obfuscated in the network. The first string represents the network path of the target ops in the original network, which should be in form of "A/B/C". The second string represents the names of multiple target ops in the same path, which should be in form of "D|E|F". For example, the target_modules of GPT2 can be ['backbone/blocks /attention', 'dense1|dense2|dense3']. If target_modules has the third value, it should be in the format of 'obfuscate_layers:all' or 'obfuscate_layers:int', which represents the number of layers need to be obfuscated of duplicate layers (such as transformer layers or resnet blocks). Default: None.
- **obf_config** (*dict, optional*) –The configuration of model obfuscation polices. Default: None.
- **saved_path** (*str, optional*) –The directory path for saving obfuscated ckpt files. Default: './'.

- **obfuscate_scale** (*Union[float, int], optional*) –Obfuscate scale of weights. The generated random obf_ratios will be in range of (1 / obfuscate_scale, obfuscate_scale). Default: `100`.

Returns

dict[str], obf_metadata, which is the necessary data that needs to be load when running obfuscated network.

Raises

- **TypeError** –If *network* is not nn.Cell.
- **TypeError** –If *ckpt_files* is not string or *saved_path* is not string.
- **TypeError** –If *target_modules* is not list.
- **TypeError** –If target_modules's elements are not string.
- **TypeError** –If obf_config is not dict.
- **ValueError** –If *ckpt_files* is not exist or *saved_path* is not exist.
- **ValueError** –If the number of elements of *target_modules* is less than `2`.
- **ValueError** –If the first string of *target_modules* contains characters other than uppercase and lowercase letters, numbers, `'_'` and `'/'`.
- **ValueError** –If the second string of *target_modules* is empty or contains characters other than uppercase and lowercase letters, numbers, `'_'` and `'|'`.
- **ValueError** –If the third string of *target_modules* is not in the format of 'obfuscate_layers:all' or 'obfuscate_layers:int'.

Examples

```
>>> from mindspore import obfuscate_ckpt, save_checkpoint
>>> # Refer to https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> save_checkpoint(net, './test_net.ckpt')
>>> target_modules = ['', 'fc1|fc2']
>>> obfuscate_ckpt(net, './', target_modules=target_modules, saved_path='./')
```

mindspore.load_obf_params_into_net

mindspore.**load_obf_params_into_net**(*network*, *target_modules=None*, *obf_ratios=None*, *obf_config=None*, *data_parallel_num=1*, ***kwargs*)

Modify model structure according to obfuscation config and load obfuscated checkpoint into obfuscated network.

Parameters

- **network** (`nn.Cell`) –The original network that need to be obfuscated.
- **target_modules** (*list[str], optional*) –The target ops that need to be obfuscated in the network. The first string represents the network path of the target ops in the original network, which should be in form of `"A/B/C"`. The second string represents the names of multiple target ops in the same path, which should be in form of `"D|E|F"`. For example, thr target_modules of GPT2 can be `['backbone /blocks/attention', 'dense1|dense2|dense3']`. If target_modules has the third value, it should be in the format of 'obfuscate_layers:all' or 'obfuscate_layers:int', which represents the number of layers need to be obfuscated of duplicate layers (such as transformer layers or resnet blocks). Default: `None`.
- **obf_ratios** (`Tensor`, *optional*) –The obf ratios generated when execute *mindspore.obfuscate_ckpt()*. Default: `None`.

- **obf_config** (*dict, optional*) –The configuration of model obfuscation polices. Default: None.
- **data_parallel_num** (*int, optional*) –The data parallel number of parallel training. Default: 1.
- **kwargs** (*dict*) –Configuration options dictionary.
  - ignored_func_decorators (list[str]): The name list of function decorators in network's python code.
  - ignored_class_decorators (list[str]): The name list of class decorators in network's python code.

Returns

nn.Cell, new_net, which is the obfuscated network.

Raises

- **TypeError** –If *network* is not nn.Cell.
- **TypeError** –If *obf_ratios* is not Tensor.
- **TypeError** –If *target_modules* is not list.
- **TypeError** –If *obf_config* is not dict.
- **TypeError** –If target_modules's elements are not string.
- **ValueError** –If the number of elements of *target_modules* is less than 2.
- **ValueError** –If *obf_ratios* is empty Tensor.
- **ValueError** –If the first string of *target_modules* contains characters other than uppercase and lowercase letters, numbers, '_' and '/'.
- **ValueError** –If the second string of *target_modules* is empty or contains characters other than uppercase and lowercase letters, numbers, '_' and '|'.
- **ValueError** –If the third string of *target_modules* is not in the format of 'obfuscate_layers:all' or 'obfuscate_layers:int'.
- **TypeError** –If *ignored_func_decorators* is not list[str] or *ignored_class_decorators* is not list[str].

Examples

```
>>> from mindspore import obfuscate_ckpt, save_checkpoint, load_checkpoint, Tensor
>>> import mindspore.common.dtype as mstype
>>> import numpy as np
>>> # Refer to https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> save_checkpoint(net, './test_net.ckpt')
>>> target_modules = ['', 'fc1|fc2']
>>> # obfuscate ckpt files
>>> obfuscate_ckpt(net, './', target_modules=target_modules, saved_path='./')
>>> # load obf ckpt into network
>>> new_net = LeNet5()
>>> load_checkpoint('./test_net_obf.ckpt', new_net)
>>> obf_net = load_obf_params_into_net(new_net, target_modules)
```

CHAPTER

TWO

MINDSPORE.OPS

mindspore.ops provides a large number of function interfaces, including neural network functions, mathematical operation functions, Tensor operation functions, Parameter operation functions, differential functions and so on.

For more information about dynamic shape support status, please refer to Dynamic Shape Support Status of ops Interface .

The module import method is as follows:

```
from mindspore import ops
```

Compared with the previous version, the added, deleted and supported platforms change information of *mindspore.ops* operators in MindSpore, please refer to the link mindspore.ops API Interface Change.

2.1 Tensor Operation Functions

2.1.1 Tensor Creation

API Name	Description	Supported Platforms
mindspore.ops.eps	Create a tensor with the same data type and shape as input, and the element value is the minimum value that the corresponding data type can express.	Ascend GPU CPU
mindspore.ops.eye	Returns a tensor with ones on the diagonal and zeros in the rest.	Ascend GPU CPU
mindspore.ops.fill	Create a tensor filled with a specified value.	Ascend GPU CPU
mindspore.ops.full	Create a tensor filled with a specified value.	Ascend GPU CPU
mindspore.ops.full_like	Return a tensor of the same shape as *input* and filled with a specified value.	Ascend GPU CPU
mindspore.ops.linspace	Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].	Ascend GPU CPU
mindspore.ops.logspace	Return a tensor with *steps* elements, evenly distributed in the interval [$base^{start}$, $base^{end}$].	Ascend GPU CPU
mindspore.ops.move_to	Copy tensor to target device synchronously or asynchronously, default synchronously.	Ascend CPU
mindspore.ops.one_hot	Generate a new tensor, where the positions specified by *indices* are assigned *on_value*, and all other positions are assigned *off_value*.	Ascend GPU CPU
mindspore.ops.ones	Creates a tensor filled with value ones.	Ascend GPU CPU
mindspore.ops.ones_like	Return a tensor filled with 1, with the same size as *input*.	Ascend GPU CPU

continues on next page

Table 1 – continued from previous page

mindspore.ops.arange	Returns a tensor with a step length of *step* in the interval [*start* , *end*).	Ascend GPU CPU
mindspore.ops.range	Returns a tensor with a step length of *step* in the interval [*start* , *end*).	GPU CPU
mindspore.ops.zeros	Creates a tensor filled with value zeros.	Ascend GPU CPU
mindspore.ops.zeros_like	Return a tensor filled with 0, with the same size as *input* .	Ascend GPU CPU
mindspore.ops.heaviside	Perform Heaviside step function element-wise.	Ascend GPU CPU

mindspore.ops.eps

mindspore.ops.**eps**(*x*)

Create a tensor with the same data type and shape as input, and the element value is the minimum value that the corresponding data type can express.

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([4, 1, 2, 3], mindspore.float32)
>>> mindspore.ops.eps(x)
Tensor(shape=[4], dtype=Float32, value= [ 1.19209290e-07,  1.19209290e-07,  1.19209290e-07,
↪1.19209290e-07])
```

mindspore.ops.eye

mindspore.ops.**eye**(*n*, *m=None*, *dtype=None*)

Returns a tensor with ones on the diagonal and zeros in the rest.

Parameters

- **n** (*int*) – The number of rows returned.
- **m** (*int, optional*) – The number of columns returned. If None , the number of columns is as the same as n.
- **dtype** (mindspore.dtype, *optional*) – The data type returned.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.eye(3)
>>> print(output)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>>
>>> output = mindspore.ops.eye(3, 4)
>>> print(output)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
```

mindspore.ops.fill

mindspore.ops.**fill**(*type*, *shape*, *value*)

Create a tensor filled with a specified value.

Parameters

- **type** (mindspore.dtype) – The data type specified.
- **shape** (*Union(Tensor, tuple[int])*) – The shape specified.
- **value** (*Union(Tensor, number.Number, bool)*) – The value specified.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.fill(mindspore.float32, (2, 3),  1.2)
Tensor(shape=[2, 3], dtype=Float32, value=
[[ 1.20000005e+00,  1.20000005e+00,  1.20000005e+00],
 [ 1.20000005e+00,  1.20000005e+00,  1.20000005e+00]])
```

mindspore.ops.full

mindspore.ops.**full**(*size*, *fill_value*, *, *dtype=None*)

Create a tensor filled with a specified value.

Note: *fill_value* 's data type not support complex numbers.

Parameters

- **size** (*Union(tuple[int], list[int])*) – The shape specified.

- **fill_value** (*number.Number*) –The value specified.

Keyword Arguments

dtype (mindspore.dtype) –The data type specified. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.full((2, 3), 1.2)
Tensor(shape=[2, 3], dtype=Int64, value=
[[1, 1, 1],
 [1, 1, 1]])
>>> mindspore.ops.full((2, 3), 1.2, dtype=mindspore.float32)
Tensor(shape=[2, 3], dtype=Float32, value=
[[ 1.20000005e+00,  1.20000005e+00,  1.20000005e+00],
 [ 1.20000005e+00,  1.20000005e+00,  1.20000005e+00]])
```

mindspore.ops.full_like

mindspore.ops.**full_like**(*input*, *fill_value*, *, *dtype=None*)

Return a tensor of the same shape as *input* and filled with a specified value.

Note: *fill_value* 's data type not support complex numbers.

Parameters

- **input** (Tensor) –The input tensor.
- **fill_value** (*Number*) –The specified value.

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default: None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.arange(4)
>>> mindspore.ops.full_like(input, 1.2)
Tensor(shape=[4], dtype=Int64, value= [1, 1, 1, 1])
>>> mindspore.ops.full_like(input, 1.2, dtype=mindspore.float32)
Tensor(shape=[4], dtype=Float32, value= [ 1.20000005e+00,  1.20000005e+00,  1.20000005e+00,
↪1.20000005e+00])
```

mindspore.ops.linspace

mindspore.ops.**linspace**(*start*, *end*, *steps*)

Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].

$$step = (end - start)/(steps - 1)$$
$$output = [start, start + step, start + 2 * step, ..., end]$$

Parameters

- **start** (*Union[Tensor, int, float]*) – Start value of interval.
- **end** (*Union[Tensor, int, float]*) – Last value of interval.
- **steps** (*Union[Tensor, int]*) – Number of elements.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.linspace(3, 10, 5)
>>> print(output)
[ 3.    4.75  6.5   8.25 10.  ]
>>> output = mindspore.ops.linspace(-10, 10, 5)
>>> print(output)
[-10.  -5.   0.   5.  10.]
>>> output = mindspore.ops.linspace(-10, 10, 1)
>>> print(output)
[-10.]
```

mindspore.ops.logspace

mindspore.ops.**logspace**(*start*, *end*, *steps*, *base=10*, ***, *dtype=mstype.float32*)

Return a tensor with *steps* elements, evenly distributed in the interval [$base^{start}$, $base^{end}$].

$$step = (end - start)/(steps - 1)$$
$$output = [base^{start}, base^{start+1*step}, ..., base^{start+(steps-2)*step}, base^{end}]$$

Parameters

- **start** (*Union[float, Tensor]*) –Start value of interval.
- **end** (*Union[float, Tensor]*) –Last value of interval.
- **steps** (*int*) –Number of elements.
- **base** (*int, optional*) –Base of the logarithm function. Default `10` .

Keyword Arguments

dtype (`mindspore.dtype`, *optional*) –The data type specified. Default `mstype.float32` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> output = mindspore.ops.logspace(1., 10., steps = 10, base = 10)
>>> print(output)
[1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.e+10]
```

mindspore.ops.move_to

mindspore.ops.**move_to**(*input*, *to='CPU'*, *blocking=True*)

Copy tensor to target device synchronously or asynchronously, default synchronously.

Note: This interface currently only supports Graph mode with jit_level of O0 or O1.

Parameters

- **input** (*Union[Tensor, list[int], tuple[int]]*) –The input tensor. When the input is list and tuple, it will be converted to tensor before copying.
- **to** (*str, optional*) –Specify the target device, with optional values of `"Ascend"` and `"CPU"`. Default `"CPU"` .
- **blocking** (*bool, optional*) –Whether use synchronous copying. Default `True`.

Returns

A new tensor on target device.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn, ops, Tensor
>>> mindspore.set_context(mode=mindspore.GRAPH_MODE)
>>> class MoveToNet(nn.Cell):
...     def __init__(self):
...         super().__init__()
...
...     def construct(self, x):
...         cpu_x = ops.move_to(x, "CPU")
...         npu_x = ops.move_to(cpu_x, "Ascend")
...         return npu_x
...
>>> net = MoveToNet()
>>> x = Tensor([1, 2, 3], mindspore.int64)
>>> y = net(x)
>>> print(y)
[1 2 3]
```

mindspore.ops.one_hot

`mindspore.ops.`**`one_hot`**(*indices*, *depth*, *on_value=1*, *off_value=0*, *axis=- 1*)

Generate a new tensor, where the positions specified by *indices* are assigned *on_value*, and all other positions are assigned *off_value*.

Note: If the input *indices* has rank *N*, the output will have rank *N+1*. The new axis is created at dimension *axis*. On Ascend, if *on_value* is int64 dtype, *indices* must be int64 dtype, and the value for *on_value* and *off_value* can only be 1 and 0.

Parameters

- **`indices`** (`Tensor`) –The input tensor of indices.
- **`depth`** (*`int`*) –The depth of the one-hot.
- **`on_value`** (*`Union[Tensor, int, float], optional`*) –The value used to fill indexed positions. Default `1` .
- **`off_value`** (*`Union[Tensor, int, float], optional`*) –The value used to fill non-indexed positions. Default `0` .
- **`axis`** (*`int, optional`*) –Specify the axis for computation. Default `-1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> indices = mindspore.tensor([0, 1, 2, 4])
>>> mindspore.ops.one_hot(indices, depth=5)
Tensor(shape=[4, 5], dtype=Int64, value=
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 0, 1]])
>>>
>>> mindspore.ops.one_hot(indices, depth=3)
Tensor(shape=[4, 3], dtype=Int64, value=
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1],
 [0, 0, 0]])
>>> # If shape of indices is (N, C), and axis=-1, the returned shape will be (N, C, depth).
>>> indices = mindspore.tensor([[0, 2], [1, -1]])
>>> mindspore.ops.one_hot(indices, depth=3, on_value=10, off_value=4, axis=-1)
Tensor(shape=[2, 2, 3], dtype=Int64, value=
[[[10,  4,  4],
  [ 4,  4, 10]],
 [[ 4, 10,  4],
  [ 4,  4,  4]]])
>>> # If axis=0, the returned shape will be (depth, N, C).
>>> mindspore.ops.one_hot(indices, depth=3, on_value=10, off_value=4, axis=0)
Tensor(shape=[3, 2, 2], dtype=Int64, value=
[[[10,  4],
  [ 4,  4]],
 [[ 4,  4],
  [10,  4]],
 [[ 4, 10],
  [ 4,  4]]])
```

mindspore.ops.ones

mindspore.ops.**ones**(*shape*, *dtype=None*)

Creates a tensor filled with value ones.

> **Warning:** For argument *shape*, Tensor type input will be deprecated in the future version.

Parameters

- **shape** (*Union[tuple[int], list[int], int, Tensor]*) – The shape specified.
- **dtype** (*mindspore.dtype*) – The data type specified. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> mindspore.ops.ones(4)
Tensor(shape=[4], dtype=Float32, value= [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00, ␣
↪1.00000000e+00])
>>> mindspore.ops.ones((2, 3))
Tensor(shape=[2, 3], dtype=Float32, value=
[[ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00]])
>>> mindspore.ops.ones(mindspore.tensor([1, 2, 3]))
Tensor(shape=[1, 2, 3], dtype=Float32, value=
[[[ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00],
  [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00]]])
```

mindspore.ops.ones_like

mindspore.ops.**ones_like**(*input*, *, *dtype=None*)

Return a tensor filled with 1, with the same size as *input*.

Parameters

input (Tensor) –The input tensor.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type specified. Default None represents the same data type as the input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.arange(4)
>>> mindspore.ops.ones_like(input)
Tensor(shape=[4], dtype=Int64, value= [1, 1, 1, 1])
```

mindspore.ops.arange

mindspore.ops.**arange**(*start=0*, *end=None*, *step=1*, *, *dtype=None*)

Returns a tensor with a step length of *step* in the interval [*start* , *end*).

Parameters

- **start** (*Union[float, int, Tensor], optional*) –The start value of the interval. Default 0 .
- **end** (*Union[float, int, Tensor], optional*) –The end value of the interval. Default None represents to the value of *start* , and *0* is used as the start value.
- **step** (*Union[float, int, Tensor], optional*) –The interval between each value. Default 1 .

Keyword Arguments

dtype (*mindspore.dtype, optional*) –The data type specified. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> mindspore.ops.arange(4)
Tensor(shape=[4], dtype=Int64, value= [0, 1, 2, 3])
>>> mindspore.ops.arange(1, 6)
Tensor(shape=[5], dtype=Int64, value= [1, 2, 3, 4, 5])
>>> mindspore.ops.arange(0, 2, 0.5)
Tensor(shape=[4], dtype=Float32, value= [ 0.00000000e+00,  5.00000000e-01,  1.00000000e+00,
 ↪1.50000000e+00])
```

mindspore.ops.range

`mindspore.ops.`**`range`**(*start*, *end*, *step*, *maxlen=1000000*)

Returns a tensor with a step length of *step* in the interval [*start* , *end*).

Note: The types of all 3 inputs must be all integers or floating-point numbers.

Parameters

- **start** (*number*) –The start value of the interval.
- **end** (*number*) –The end value of the interval.
- **step** (*number*) –The interval between each value.
- **maxlen** (*int, optional*) –Memory that can fit *maxlen* many elements will be allocated for the output. Optional, must be positive. Default: 1000000. If the output has more than *maxlen* elements, a runtime error will occur.

Returns

Tensor

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> mindspore.ops.range(0, 6, 2)
Tensor(shape=[3], dtype=Int64, value= [0, 2, 4])
```

mindspore.ops.zeros

mindspore.ops.**zeros**(*size*, *dtype=None*)

Creates a tensor filled with value zeros.

> **Warning:** For argument *size*, Tensor type input will be deprecated in the future version.

Parameters

- **size** (*Union[tuple[int], list[int], int, Tensor]*) −The shape specified.
- **dtype** (*mindspore.dtype*, optional) −The data type specified. Default: None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.zeros(4)
Tensor(shape=[4], dtype=Float32, value= [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ␣
↪0.00000000e+00])
>>> mindspore.ops.zeros((2, 3))
Tensor(shape=[2, 3], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
 [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
>>> mindspore.ops.zeros(mindspore.tensor([1, 2, 3]))
Tensor(shape=[1, 2, 3], dtype=Float32, value=
[[[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
  [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]]])
```

mindspore.ops.zeros_like

mindspore.ops.**zeros_like**(*input*, *, *dtype=None*)

Return a tensor filled with 0, with the same size as *input* .

Parameters

input (Tensor) −The input tensor.

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The data type specified. Default None represents the same data type as the input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.ops.arange(4)
>>> mindspore.ops.zeros_like(input)
Tensor(shape=[4], dtype=Int64, value= [0, 0, 0, 0])
```

mindspore.ops.heaviside

`mindspore.ops.`**`heaviside`**(*input*, *values*)

Perform Heaviside step function element-wise.

Support broadcasting.

$$\text{heaviside}(\text{input}, \text{values}) = \begin{cases} 0, & \text{if input} < 0 \\ \text{values}, & \text{if input} = 0 \\ 1, & \text{if input} > 0 \end{cases}$$

Parameters

- **`input`** (`Tensor`) –The input tensor.
- **`values`** (`Tensor`) –The value to fill when the element in *input* is 0.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-2., 0, 3],
...                           [5, -1, 0],
...                           [0, 7, -3]])
>>> values = mindspore.tensor([2, 0.5, 1])
>>> output = mindspore.ops.heaviside(input, values)
>>> print(output)
[[0.  0.5 1. ]
 [1.  0.  1. ]
 [2.  1.  0. ]]
>>> output = mindspore.ops.heaviside(input, mindspore.tensor(0.5))
>>> print(output)
[[0.  0.5 1. ]
 [1.  0.  0.5]
```

(continues on next page)

(continued from previous page)

```
 [0.5 1.  0. ]]
>>> output = mindspore.ops.heaviside(mindspore.tensor(-3.), values)
>>> print(output)
[0. 0. 0.]
```

2.1.2 Randomly Generating Functions

API Name	Description	Supported Platforms
mindspore.ops.bernoulli	Generates Bernoulli random values (0 or 1).	GPU CPU
mindspore.ops.gamma	Generates random numbers according to the Gamma random number distribution.	Ascend
mindspore.ops.laplace	Generates random numbers according to the Laplace random number distribution.	Ascend GPU CPU
mindspore.ops.multinomial	Generate a tensor from a multinomial distribution.	Ascend GPU CPU
mindspore.ops.multinomial_with_replacement	Generate a tensor from a multinomial distribution.	CPU
mindspore.ops.rand	Return a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given *size* and *dtype*.	Ascend GPU CPU
mindspore.ops.rand_like	Return a tensor with the same shape as *input* that is filled with random numbers from a uniform distribution on the interval $[0, 1)$.	Ascend GPU CPU
mindspore.ops.randint	Return a tensor whose elements are random integers in the range of [*low* , *high*) .	Ascend GPU CPU
mindspore.ops.randint_like	Returns a tensor with the same shape as *input* whose elements are random integers in the range of [*low* , *high*) .	Ascend GPU CPU
mindspore.ops.randn	Return a new tensor with given shape and dtype, filled with random numbers from the standard normal distribution.	Ascend GPU CPU
mindspore.ops.randn_like	Return a tensor with the same shape as *input*, filled with random numbers from the standard normal distribution.	Ascend GPU CPU
mindspore.ops.random_gamma	Generate random numbers from the Gamma distribution(s).	CPU
mindspore.ops.random_poisson	Generate random number Tensor with *shape* according to a Poisson distribution with mean *rate*.	GPU CPU
mindspore.ops.randperm	Generates random permutation of integers from 0 to n-1.	CPU
mindspore.ops.standard_laplace	Generates random numbers according to the Laplace random number distribution (mean=0, lambda=1).	Ascend GPU CPU
mindspore.ops.standard_normal	Generates random numbers according to the standard Normal (or Gaussian) random number distribution.	Ascend GPU CPU
mindspore.ops.uniform	Generates random numbers according to the Uniform random number distribution.	GPU CPU

mindspore.ops.bernoulli

mindspore.ops.**bernoulli**(*input*, *p=0.5*, *seed=None*)

Generates Bernoulli random values (0 or 1).

$$out_i \sim Bernoulli(p_i)$$

Parameters

- **input** (Tensor) –The input Tensor.
- **p** (*Union[Tensor, float], optional*) –The probability of setting 1 for the corresponding position of the returned tensor. The value of *p* must be in the range *[0, 1]*. Default 0.5 .
- **seed** (*Union[int, None], optional*) –The random seed. Default None means using the current timestamp.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3])
>>> mindspore.ops.bernoulli(input, p=1.0)
Tensor(shape=[3], dtype=Int64, value= [1, 1, 1])
>>> p = mindspore.tensor([0.0, 1.0, 1.0])
>>> mindspore.ops.bernoulli(input, p)
Tensor(shape=[3], dtype=Int64, value= [0, 1, 1])
```

mindspore.ops.gamma

mindspore.ops.**gamma**(*shape*, *alpha*, *beta*, *seed=None*)

Generates random numbers according to the Gamma random number distribution.

Support broadcasting.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (*tuple*) –The shape specified.
- **alpha** (Tensor) –The shape parameter.
- **beta** (Tensor) –The inverse scale parameter.
- **seed** (*int, optional*) –The random seed, Default None .

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1: alpha_shape is (2, 2)
>>> shape = (3, 1, 2)
>>> alpha = mindspore.tensor([[3, 4], [5, 6]], mindspore.float32)
>>> beta = mindspore.tensor([1.0], mindspore.float32)
>>> output = mindspore.ops.gamma(shape, alpha, beta, seed=5)
>>> result = output.shape
>>> print(result)
(3, 2, 2)
>>> # case 2: alpha_shape is (2, 3), so shape is (3, 1, 3)
>>> shape = (3, 1, 3)
>>> alpha = mindspore.tensor([[1, 3, 4], [2, 5, 6]]), mindspore.float32)
>>> beta = mindspore.tensor([1.0], mindspore.float32)
>>> output = mindspore.ops.gamma(shape, alpha, beta, seed=5)
>>> result = output.shape
>>> print(result)
(3, 2, 3)
>>> # case 3: beta_shape is (1, 2), the output is different.
>>> shape = (3, 1, 2)
>>> alpha = mindspore.tensor([[3, 4], [5, 6]], mindspore.float32)
>>> beta = mindspore.tensor([1.0, 2], mindspore.float32)
>>> output = mindspore.ops.gamma(shape, alpha, beta, seed=5)
>>> print(output)
[[[ 2.2132034  5.8855834]
  [ 3.8825176  8.6066265]]
 [[ 3.3981476  7.5805717]
  [ 3.7190282 19.941492 ]]
 [[ 2.9512358  2.5969937]
  [ 3.786061   5.160872 ]]]
>>> # case 4: beta_shape is (2, 1), the output is different.
>>> shape = (3, 1, 2)
>>> alpha = mindspore.tensor([[3, 4], [5, 6]], mindspore.float32)
>>> beta = mindspore.tensor([[1.0], [2.0]], mindspore.float32)
>>> output = mindspore.ops.gamma(shape, alpha, beta, seed=5)
>>> print(output)
[[[ 5.6085486  7.8280783]
  [ 15.97684  16.116285]]
[[ 1.8347423  1.713663]
  [ 3.2434065 15.667398]]
[[ 4.2922077  7.3365674]
  [ 5.3876944  13.159832 ]]]
```

mindspore.ops.laplace

mindspore.ops.**laplace**(*shape*, *mean*, *lambda_param*, *seed=None*)

Generates random numbers according to the Laplace random number distribution.

Support broadcasting.

$$\mathrm{f}(x;,) = \frac{1}{2}\exp(-\frac{|x-|}{}),$$

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (*tuple*) –The shape specified.
- **mean** (Tensor) –The mean of distribution.
- **lambda_param** (Tensor) –Control the variance of distribution. The variance of Laplace distribution is equal to twice the square of *lambda_param* .
- **seed** (*int, optional*) –Random seed. Default None represents 0.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> shape = (2, 3)
>>> mean = mindspore.tensor(1.0, mindspore.float32)
>>> lambda_param = mindspore.tensor(1.0, mindspore.float32)
>>> output = mindspore.ops.laplace(shape, mean, lambda_param, seed=5)
>>> print(output.shape)
(2, 3)
```

mindspore.ops.multinomial

mindspore.ops.**multinomial**(*input*, *num_samples*, *replacement=True*, *seed=None*)

Generate a tensor from a multinomial distribution.

The polynomial distribution is a probability distribution that generalizes the binomial distribution formula to multiple states. In the polynomial distribution, each event has a fixed probability, and the sum of these probabilities is 1.

The purpose of this interface is to perform *num_samples* sampling on the input *input*, and the output tensor is the index of the input tensor for each sampling. The values in *input* represent the probability of selecting the corresponding index for each sampling.

Here is an extreme example for better understanding. Suppose we have an input probability tensor with values *[90 / 100, 10 / 100, 0]*, which means we can sample three indices, namely index 0, index 1, and index 2, with probabilities of 90%, 10%, and 0%, respectively. We perform n samplings, and the resulting sequence is the calculation result of the polynomial distribution, with a length equal to the number of samplings.

In case 1 of the sample code, we perform two non-replacement samplings (*replacement* is *False*). Since the probability of selecting index 0 is 90% for each sampling, the first result is most likely to be index 0. Since the probability of selecting index 2 is 0, index 2 cannot appear in the sampling result. Therefore, the second result must be index 1, and the resulting sequence is *[0, 1]*.

In case 2 of the sample code, we perform 10 replacement samplings (*replacement* is *True*). As expected, about 90% of the sampling results are index 0.

In case 3 of the sample code, we extend the input to 2 dimensions, and the sampling results in each dimension also match our sampling expectations.

Note: The rows of input do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum. When using values as weights, it can be understood as normalizing the input along the last dimension.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input** (Tensor) –The input tensor containing probabilities.
- **num_samples** (*int*) –Number of samples to draw.
- **replacement** (*bool, optional*) –Whether to draw with replacement or not. Default `True` .
- **seed** (*int, optional*) –Random seed. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is False.
>>> input1 = mindspore.tensor([90 / 100, 10 / 100, 0])
>>> input2 = mindspore.tensor([90, 10, 0])
>>> # input1 and input2 have the same meaning.
>>> mindspore.ops.multinomial(input1, 2, replacement=False)
Tensor(shape=[2], dtype=Int32, value= [0, 1])
>>> mindspore.ops.multinomial(input2, 2, replacement=False)
Tensor(shape=[2], dtype=Int32, value= [1, 0])
>>>
>>> # case 2: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is True.
>>> mindspore.ops.multinomial(input1, 10)
Tensor(shape=[10], dtype=Int32, value= [0, 0, 1, 0, 0, 0, 0, 0, 0, 0])
>>>
>>> # case 3: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is True.
>>> # rank is 2
>>> input3 = mindspore.tensor([[90, 10, 0], [10, 90, 0]], mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> output = mindspore.ops.multinomial(input3, 10)
>>> print(output)
[[0 0 0 0 0 0 0 0 0 0]
 [1 0 1 1 1 1 1 1 1 1]]
```

mindspore.ops.multinomial_with_replacement

mindspore.ops.**multinomial_with_replacement**(*x*, *seed*, *offset*, *numsamples*, *replacement=False*)

Generate a tensor from a multinomial distribution.

Note:

- The rows of input do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum.
- If *seed* is set to be `-1` , and *offset* is set to be `0`, the random number generator is seeded by a random seed.

Parameters

- **x** (*Tensor*) – The 1-D or 2-D input tensor containing probabilities.
- **seed** (*int*) – Random seed.
- **offset** (*int*) – Offset.
- **numsamples** (*int*) – The number of samples to draw.
- **replacement** (*bool, optional*) – Whether to draw with replacement or not. Default `False` .

Returns

Tensor

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[0., 9., 4., 0.]], mindspore.float32)
>>> mindspore.ops.multinomial_with_replacement(x, 2, 5, 2, True)
Tensor(shape=[1, 2], dtype=Int64, value=
[[1, 1]])
```

mindspore.ops.rand

mindspore.ops.**rand**(**size*, *dtype=None*, *seed=None*)

Return a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given *size* and *dtype*.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

size (*Union[int, tuple(int), list(int)]*) – The shape of the output tensor.

Keyword Arguments

- **dtype** (*mindspore.dtype*, optional) – The data type returned. Default `None` .
- **seed** (*int, optional*) – Random seed, must be greater or equal to 0. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> print(mindspore.ops.rand((2,3)))
[[4.1702199e-01 9.9718481e-01 7.2032452e-01]
 [9.3255734e-01 1.1438108e-04 1.2812445e-01]]
```

mindspore.ops.rand_like

mindspore.ops.**rand_like**(*input*, *seed=None*, ***, *dtype=None*)

Return a tensor with the same shape as *input* that is filled with random numbers from a uniform distribution on the interval $[0, 1)$.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input** (`Tensor`) – The input tensor.
- **seed** (*int, optional*) – Random seed, must be greater or equal to 0. Default `None` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The data type returned. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([[2, 3, 4], [1, 2, 3]])
>>> print(mindspore.ops.rand_like(a, dtype=mindspore.float32))
[[4.1702199e-01 9.9718481e-01 7.2032452e-01]
 [9.3255734e-01 1.1438108e-04 1.2812445e-01]]
```

mindspore.ops.randint

mindspore.ops.**randint**(*low*, *high*, *size*, *seed=None*, ***, *dtype=None*)

Return a tensor whose elements are random integers in the range of [*low* , *high*) .

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **low** (*int*) –Start value of interval.
- **high** (*int*) –End value of interval.
- **size** (*tuple*) –Shape of the output tensor.
- **seed** (*int, optional*) –Random seed, must be non-negative. Default `None` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> print(mindspore.ops.randint(1, 10, (2,3)))
[[4 9 7]
 [9 1 2]]
```

mindspore.ops.randint_like

mindspore.ops.**randint_like**(*input*, *low*, *high*, *seed=None*, ***, *dtype=None*)

Returns a tensor with the same shape as *input* whose elements are random integers in the range of [*low* , *high*) .

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input** (Tensor) –The input tensor.
- **low** (*int*) –Start value of interval.
- **high** (*int*) –End value of interval.
- **seed** (*int, optional*) –Random seed, must be non-negative. Default None .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([[1, 2, 3], [3, 2, 1]])
>>> print(mindspore.ops.randint_like(a, 1, 10))
[[4 9 7]
 [9 1 2]]
```

mindspore.ops.randn

mindspore.ops.**randn**(**size*, *dtype=None*, *seed=None*)

Return a new tensor with given shape and dtype, filled with random numbers from the standard normal distribution.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

size (*Union[int, tuple(int), list(int)]*) –Shape of the output tensor.

Keyword Arguments

- **dtype** (*mindspore.dtype*, optional) –The data type returned. Default None .
- **seed** (*int, optional*) –Random seed, must be non-negative. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> print(mindspore.ops.randn((2, 2)))
[[ 0.30639967 -0.42438635]
 [-0.4287376   1.3054721 ]]
```

mindspore.ops.randn_like

mindspore.ops.**randn_like**(*input*, *seed=None*, *, *dtype=None*)

Return a tensor with the same shape as *input*, filled with random numbers from the standard normal distribution.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input** (Tensor) –The input tensor.
- **seed** (*int, optional*) –Random seed, must be non-negative. Default None .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([[1, 2, 3], [4, 5, 6]])
>>> print(mindspore.ops.randn_like(a, dtype=mindspore.float32))
[[ 0.30639967 -0.42438635 -0.20454668]
 [-0.4287376   1.3054721   0.64747655]]
```

mindspore.ops.random_gamma

mindspore.ops.**random_gamma**(*shape*, *alpha*, *seed=None*)

Generate random numbers from the Gamma distribution(s).

Parameters

- **shape** (Tensor) –The shape of random tensor to be generated.
- **alpha** (Tensor) –The α distribution parameter.
- **seed** (*int, optional*) –Random seed, must be non-negative. Default None .

Returns

Tensor, the shape is *mindspore.ops.concat([shape, rate.shape], axis=0)*. The data type is the same as *alpha*.

Supported Platforms:

CPU

Examples

```
>>> import mindspore
>>> shape = mindspore.tensor([7, 5], mindspore.int32)
>>> alpha = mindspore.tensor([0.5, 1.5], mindspore.float32)
>>> output = mindspore.ops.random_gamma(shape, alpha, seed=5)
>>> print(output.shape, output.dtype)
(7, 5, 2) Float32
```

mindspore.ops.random_poisson

mindspore.ops.**random_poisson**(*shape*, *rate*, *seed=None*, *dtype=mstype.float32*)

Generate random number Tensor with *shape* according to a Poisson distribution with mean *rate*.

$$\mathrm{P}(i|) = \frac{\exp(-)^i}{i!}$$

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (Tensor) –The shape of random tensor to be sampled from each poisson distribution, 1-D integer tensor.
- **rate** (Tensor) –The parameter the distribution is constructed with. It represents the mean of poisson distribution and also the variance of the distribution.
- **seed** (*int, optional*) –Random seed, must be non-negative. Default None .
- **dtype** (mindspore.dtype) –The data type returned. Default mstype.float32.

Returns

Tensor, the shape is *mindspore.ops.concat([shape, rate.shape], axis=0)*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: 1-D shape, 2-D rate, float64 output
>>> shape = mindspore.tensor([2, 2], mindspore.int64)
>>> rate = mindspore.tensor([[5.0, 10.0], [5.0, 1.0]], mindspore.float32)
>>> output = mindspore.ops.random_poisson(shape, rate, seed=5, dtype=mindspore.float64)
>>> print(output.shape, output.dtype)
(2, 2, 2, 2) Float64
>>> # case 2: 1-D shape, scalar rate, int64 output
>>> shape = mindspore.tensor([2, 2], mindspore.int64)
```

(continues on next page)

(continued from previous page)

```
>>> rate = mindspore.tensor(5.0, mindspore.float64)
>>> output = mindspore.ops.random_poisson(shape, rate, seed=5, dtype=mindspore.int64)
>>> print(output.shape, output.dtype)
(2, 2) Int64
```

mindspore.ops.randperm

mindspore.ops.**randperm**(*n*, *seed=0*, *offset=0*, *dtype=mstype.int64*)

Generates random permutation of integers from 0 to n-1.

Warning:

- This is an experimental API that is subject to change or deletion.
- The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **n** (*Union[Tensor, int]*) – The upper bound (exclusive).
- **seed** (*int, optional*) – Random seed. Default `0` . When seed is -1, offset is 0, it's determined by time.
- **offset** (*int, optional*) – Offset to generate random numbers. Priority is higher than random seed. Default `0` . It must be non-negative.
- **dtype** (mindspore.dtype, *optional*) – The data type returned. Default `mstype.int64`.

Returns

Tensor. Its shape is specified by the required args *n*.

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> n, seed, offset = 4, 0, 0
>>> output = mindspore.ops.randperm(n, seed, offset, dtype=mindspore.int64)
>>> print(output)
[0 2 1 3]
```

mindspore.ops.standard_laplace

mindspore.ops.**standard_laplace**(*shape*, *seed=None*)

Generates random numbers according to the Laplace random number distribution (mean=0, lambda=1).

$$\mathrm{f}(x) = \frac{1}{2}\exp(-|x|)$$

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (*Union[tuple, Tensor]*) –The shape of returned tensor.
- **seed** (*int, optional*) –Random number seed. Default `None` .

Returns

Tensor

Raises

- **ValueError** –If shape is a tuple containing non-positive items.
- **ValueError** –If shape is a Tensor, and the rank of the Tensor is not equal to 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> shape = (4, 4)
>>> output = mindspore.ops.standard_laplace(shape, seed=5)
>>> print(f'output shape is {output.shape}')
output shape is (4, 4)
```

mindspore.ops.standard_normal

`mindspore.ops.`**`standard_normal`**(*shape*, *seed=None*)

Generates random numbers according to the standard Normal (or Gaussian) random number distribution.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{\left(-\frac{x^2}{2}\right)}$$

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (*Union[tuple, Tensor]*) –The shape of returned tensor.
- **seed** (*int, optional*) –Random number Seed. Default `None` .

Returns

Tensor

Raises

- **ValueError** –If *shape* is a tuple containing non-positive items.
- **ValueError** –If shape is a Tensor, and the rank of the Tensor is not equal to 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> shape = (4, 4)
>>> output = mindspore.ops.standard_normal(shape, seed=5)
>>> print(f'output shape is {output.shape}')
output shape is (4, 4)
```

mindspore.ops.uniform

mindspore.ops.**uniform**(*shape*, *minval*, *maxval*, *seed=None*, *dtype=mstype.float32*)

Generates random numbers according to the Uniform random number distribution.

Note: The number in tensor minval should be strictly less than maxval at any position after broadcasting.

Parameters

- **shape** (*Union[tuple, Tensor]*) –The shape of returned tensor.
- **minval** (Tensor) –Defines the minimum possible generated value.
- **maxval** (Tensor) –Defines the maximum possible generated value.
- **seed** (*int*) –Random number seed. Default None .
- **dtype** (mindspore.dtype) –Type of the returned tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> # For discrete uniform distribution, only one number is allowed for both minval and␣
↪maxval:
>>> shape = (4, 2)
>>> minval = mindspore.tensor(1, mindspore.int32)
>>> maxval = mindspore.tensor(2, mindspore.int32)
>>> output = mindspore.ops.uniform(shape, minval, maxval, seed=5, dtype=mindspore.int32)
>>>
>>> # For continuous uniform distribution, minval and maxval can be multi-dimentional:
>>> shape = (3, 1, 2)
>>> minval = mindspore.tensor([[3, 4], [5, 6]], mindspore.float32)
>>> maxval = mindspore.tensor([8.0, 10.0], mindspore.float32)
>>> output = mindspore.ops.uniform(shape, minval, maxval, seed=5)
```

(continues on next page)

(continued from previous page)

```
>>> result = output.shape
>>> print(result)
(3, 2, 2)
```

2.1.3 Array Operation

API Name	Description	Supported Platforms
mindspore.ops.argwhere	Return a tensor of containing the positions of all non-zero elements in the input tensor.	Ascend GPU CPU
mindspore.ops.batch_to_space_nd	Divides batch dimension with blocks and interleaves these blocks back into spatial dimensions.	Ascend GPU CPU
mindspore.ops.bincount	Count the frequency of each value in the input tensor of non-negative ints.	Ascend GPU CPU
mindspore.ops.block_diag	Creates a block diagonal matrix from the provided tensor.	Ascend GPU CPU
mindspore.ops.broadcast_to	Broadcasts input tensor to a given shape.	Ascend GPU CPU
mindspore.ops.cat	Connect input tensors along with the given axis.	Ascend GPU CPU
mindspore.ops.channel_shuffle	Divide the channels in a tensor of shape $(*, C, H, W)$ into g groups and rearrange them as $(*, \frac{C}{g}, g, H*W)$, while retaining the original tensor shape in the final output.	Ascend CPU
mindspore.ops.chunk	Split the input tensor into multiple sub-tensors along the specified axis.	Ascend GPU CPU
mindspore.ops.column_stack	Creates a new tensor by horizontally stacking the tensors in *tensors*.	Ascend GPU CPU
mindspore.ops.concat	Alias for mindspore.ops.cat().	
mindspore.ops.conj	Returns a tensor of complex numbers that are the complex conjugate of each element in input.	Ascend GPU CPU
mindspore.ops.count_nonzero	Counts the number of non-zero values in the input tensor along the given axis.	Ascend GPU CPU
mindspore.ops.deepcopy	Return a deepcopy of input tensor.	Ascend GPU CPU
mindspore.ops.diag	Return a tensor with the *input* as its diagonal elements and 0 elsewhere.	Ascend GPU CPU
mindspore.ops.diagflat	If *input* is a vector (1-D tensor), then returns a 2-D square tensor with the elements of *input* as the diagonal, If *input* is a tensor with more than one dimension, then returns a 2-D tensor with diagonal elements equal to a flattened *input*.	Ascend GPU CPU
mindspore.ops.diagonal	Returns diagonals of the input tensor along specified dimension.	Ascend GPU CPU
mindspore.ops.diagonal_scatter	Embeds the values of the *src* tensor into *input* along the diagonal elements of input, with respect to *dim1* and *dim2* .	Ascend GPU CPU
mindspore.ops.diag_embed	Create a tensor whose diagonals of certain 2D planes (specified by *dim1* and *dim2*) are filled by *input*, and all other positions are set to 0.	Ascend GPU CPU
mindspore.ops.dyn_shape	Returns the shape of the input tensor.	Ascend GPU CPU
mindspore.ops.dsplit	Splits a tensor along the 3rd axis.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.ops.dstack	Stacks tensors along the third axis.	Ascend GPU CPU
mindspore.ops.einsum	According to the Einstein summation Convention (Einsum), the product of the input tensor elements is summed along the specified dimension.	GPU
mindspore.ops.expand_dims	Adds an additional axis to input tensor.	Ascend GPU CPU
mindspore.ops.flip	Reverses elements in a tensor along the given dims.	Ascend GPU CPU
mindspore.ops.fliplr	Flip the input tensor in left/right direction.	Ascend GPU CPU
mindspore.ops.flipud	Flip the input tensor in up/down direction.	Ascend GPU CPU
mindspore.ops.gather	Returns the slice of the input tensor corresponding to the elements of *input_indices* on the specified *axis*.	Ascend GPU CPU
mindspore.ops.gather_d	Gathers elements along an axis specified by dim.	Ascend GPU CPU
mindspore.ops.gather_elements	Gathers elements along the specified dim and indices.	Ascend GPU CPU
mindspore.ops.gather_nd	Gathers slices from the input tensor by specified indices.	Ascend GPU CPU
mindspore.ops.hstack	Stacks tensors in sequence horizontally.	Ascend GPU CPU
mindspore.ops.hsplit	Splits a tensor into multiple sub-tensors horizontally.	Ascend GPU CPU
mindspore.ops.index_add	Add the elements of input *y* into input *x* along the given axis and indices.	Ascend GPU CPU
mindspore.ops.index_fill	Fills the elements of the input *x* with *value* along the given axis and indices.	Ascend GPU CPU
mindspore.ops.index_select	Select the input tensor according to the specified axis and index and return a new tensor.	Ascend GPU CPU
mindspore.ops.inplace_add	Add *x* to *v* according to the indices.	Ascend GPU CPU
mindspore.ops.inplace_index_add	Add *updates* to *var* according to the indices and axis.	Ascend CPU
mindspore.ops.inplace_sub	Subtract *v* in *x* according to the indices.	Ascend GPU CPU
mindspore.ops.inplace_update	Updates *x* to *v* according to the indices.	GPU CPU
mindspore.ops.is_nonzero	Determine whether the input Tensor contains `0` or `False`.	Ascend GPU CPU
mindspore.ops.masked_fill	Fills elements of tensor with value where mask is `True`.	Ascend GPU CPU
mindspore.ops.masked_select	Return a new 1-D tensor which indexes the *input* tensor according to the boolean *mask*.	Ascend GPU CPU
mindspore.ops.meshgrid	Creates grids of coordinates specified by the 1D inputs。	Ascend GPU CPU
mindspore.ops.narrow	Slice the tensor from the *start* position with a length of *length* along *axis* .	Ascend GPU CPU
mindspore.ops.moveaxis	Move axis of an array from source to destination.	Ascend GPU CPU
mindspore.ops.movedim	Swap two dimensions of the input tensor.	Ascend GPU CPU
mindspore.ops.nan_to_num	Replace the *NaN*, positive infinity and negative infinity values in *input* with the specified values in *nan*, *posinf* and *neginf* respectively.	Ascend CPU
mindspore.ops.nanmean	Computes the mean of *input* in specified dimension, ignoring NaN.	Ascend GPU CPU
mindspore.ops.nanmedian	Computes the median and indices of *input* in specified dimension, ignoring NaN.	CPU
mindspore.ops.nansum	Computes sum of *input* over a given dimension, ignoring NaN.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.ops.normal	Return a random tensor that conforms to the normal (Gaussian) distribution.	Ascend GPU CPU
mindspore.ops.nonzero	Return the positions of all non-zero values.	Ascend GPU CPU
mindspore.ops.numel	Return the total number of elements in the tensor.	Ascend GPU CPU
mindspore.ops.permute	Permute the input tensor along the specified axis.	Ascend GPU CPU
mindspore.ops.population_count	Calculate the number of 1 bits in the binary representation of each element in the input tensor.	Ascend GPU CPU
mindspore.ops.rank	Return the rank of a tensor.	Ascend GPU CPU
mindspore.ops.repeat_elements	Repeat elements of a tensor along an axis, like mindspore.numpy.repeat() .	Ascend GPU CPU
mindspore.ops.repeat_interleave	Repeat elements of a tensor along an axis, like mindspore.numpy.repeat().	Ascend GPU CPU
mindspore.ops.reshape	Reshape the input tensor based on the given shape.	Ascend GPU CPU
mindspore.ops.reverse_sequence	Partially reverse the input sequence.	Ascend GPU CPU
mindspore.ops.roll	Roll the elements of a tensor along a dimension.	Ascend GPU
mindspore.ops.row_stack	Alias for mindspore.ops.vstack() .	Ascend GPU CPU
mindspore.ops.scatter	Update the value in *src* to *input* according to the specified index.	Ascend GPU CPU
mindspore.ops.scatter_nd	Scatters a tensor into a new tensor depending on the specified indices.	Ascend GPU CPU
mindspore.ops.select	The conditional tensor determines whether the corresponding element in the output must be selected from *input* (if True) or *other* (if False) based on the value of each element.	Ascend GPU CPU
mindspore.ops.select_scatter	On the specified dimension *axis* of *input* , *src* is scattered into *input* on the specified *index* of *input* .	Ascend GPU CPU
mindspore.ops.sequence_mask	Returns a mask tensor representing the first N positions of each cell.	GPU CPU
mindspore.ops.shape	Return the shape of the input tensor.	Ascend GPU CPU
mindspore.ops.shuffle	Randomly shuffle a tensor along its first dimension.	Ascend GPU CPU
mindspore.ops.size	Count the total number of elements in *input_x* .	Ascend GPU CPU
mindspore.ops.slice	Slice a tensor in the specified shape.	Ascend GPU CPU
mindspore.ops.slice_scatter	Embed *src* into the sliced *input* along the specified *axis* .	Ascend GPU CPU
mindspore.ops.sort	Sort the elements of the input tensor along the given axis.	Ascend GPU CPU
mindspore.ops.space_to_batch_nd	Divides a tensor's spatial dimensions into blocks and combines the block sizes with the original batch.	Ascend GPU CPU
mindspore.ops.sparse_segment_mean	Computes the mean of sparse segments in the input tensor.	GPU CPU
mindspore.ops.split	Split the tensor into chunks along the given axis.	Ascend GPU CPU
mindspore.ops.squeeze	Remove length one axes from input tensor.	Ascend GPU CPU
mindspore.ops.stack	Stack input tensors in specified axis.	Ascend GPU CPU
mindspore.ops.strided_slice	Extracts a strided slice of a Tensor based on *begin/end* index and *strides*.	Ascend GPU CPU
mindspore.ops.sum	Calculate sum of tensor elements over a given dim.	Ascend GPU CPU
mindspore.ops.swapaxes	Interchange two axes of a tensor.	Ascend GPU CPU
mindspore.ops.swapdims	Interchange two dims of a tensor.	Ascend GPU CPU
mindspore.ops.tensor_scatter_add	Return a new tensor by adding the values from *updates* in *input_x* indicated by *indices* .	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.ops.tensor_scatter_div	Return a new tensor which *input_x* is divided by the values from *updates* indicated by *indices* .	GPU CPU
mindspore.ops.tensor_scatter_max	Return a new tensor by performing a maximum update on *input_x* at the specified indices with the given update values.	GPU CPU
mindspore.ops.tensor_scatter_min	Return a new tensor by performing a minimum update on *input_x* at the specified indices with the given update values.	Ascend GPU CPU
mindspore.ops.tensor_scatter_mul	Return a new tensor by performing a multiplication update on *input_x* at the specified indices with the given update values.	GPU CPU
mindspore.ops.tensor_scatter_sub	Return a new tensor by performing a subtraction update on *input_x* at the specified indices with the given update values.	Ascend GPU CPU
mindspore.ops.tensor_scatter_elements	Return a new tensor by performing a specified operation update on *input_x* at the specified indices with the given update values.	Ascend GPU CPU
mindspore.ops.tensor_split	Split the input tensor into multiple subtensors according to the specified indices or chunks.	Ascend GPU CPU
mindspore.ops.tile	Creates a new tensor by repeating the elements in the input tensor *dims* times.	Ascend GPU CPU
mindspore.ops.tril	Zero the input tensor above the diagonal specified.	Ascend GPU CPU
mindspore.ops.triu	Zero the input tensor below the diagonal specified.	Ascend GPU CPU
mindspore.ops.transpose	Transpose dimensions of the input tensor according to input permutation.	Ascend GPU CPU
mindspore.ops.unbind	Remove a tensor dimension, return a tuple of all slices along a given dimension.	Ascend GPU CPU
mindspore.ops.unique	Remove duplicate elements from the input tensor.	Ascend GPU CPU
mindspore.ops.unique_consecutive	Remove consecutive duplicate elements in the input tensor, retaining only the first occurrence from each repeated group.	Ascend GPU CPU
mindspore.ops.unsorted_segment_max	Compute the maximum of the input tensor along segments.	Ascend GPU CPU
mindspore.ops.unsorted_segment_min	Compute the minimum of the input tensor along segments.	Ascend GPU CPU
mindspore.ops.unsorted_segment_prod	Compute the product of the input tensor along segments.	Ascend GPU CPU
mindspore.ops.unsorted_segment_sum	Compute the sum of the input tensor along segments.	Ascend GPU CPU
mindspore.ops.unsqueeze	Adds an additional dimension to the input tensor at the given dimension.	Ascend GPU CPU
mindspore.ops.unstack	Unstack the input tensor along the specified axis.	Ascend GPU CPU
mindspore.ops.view_as_real	Return a real tensor with the last dimension of size 2, composed of the real and imaginary parts of the complex elements in the input tensor.	GPU CPU
mindspore.ops.vsplit	Split the input tensor with two or more dimensions, into multiple sub-tensors vertically according to *indices_or_sections*.	Ascend GPU CPU
mindspore.ops.vstack	Stacks tensors in sequence vertically.	Ascend GPU CPU
mindspore.ops.where	Return a tensor in which the elements are selected from *input* or *other* based on the *condition*.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.ops.cross	Compute the cross product of two input tensors along the specified dimension.	Ascend CPU
mindspore.ops.renorm	Returns a tensor where each subtensor along the specified dimension is renormalized such that its p norm is less than or equal to *maxnorm*.	Ascend GPU CPU

mindspore.ops.argwhere

mindspore.ops.**argwhere**(*input*)

Return a tensor of containing the positions of all non-zero elements in the input tensor.

Parameters

input (Tensor) –The input tensor.

Returns

2-D Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[[1,  0], [-5, 0]]], mindspore.int32)
>>> output = mindspore.ops.argwhere(x)
>>> print(output)
[[0 0 0]
 [0 1 0]]
```

mindspore.ops.batch_to_space_nd

mindspore.ops.**batch_to_space_nd**(*input_x*, *block_shape*, *crops*)

Divides batch dimension with blocks and interleaves these blocks back into spatial dimensions.

This operation will divide batch dimension N into blocks with block_shape, the output tensor's N dimension is the corresponding number of blocks after division. The output tensor's $w_1, ..., w_M$ dimension is the product of original $w_1, ..., w_M$ dimension and block_shape with given amount to crop from dimension, respectively.

If the input shape is $(n, c_1, ...c_k, w_1, ..., w_M)$, the output shape is $(n', c_1, ...c_k, w'_1, ..., w'_M)$, where

$$n' = n//(block\_shape[0] * ... * block\_shape[M-1])$$
$$w'_i = w_i * block\_shape[i-1] - crops[i-1][0] - crops[i-1][1]$$

Parameters

- **input_x** (Tensor) –The input tensor. It must be greater or equal to 2-D tensor(equal to 4-D tensor on Ascend), batch dimension must be divisible by product of *block_shape*.
- **block_shape** (*Union[list(int), tuple(int), int]*) –The block shape of dividing block with all value greater than or equal to 1. If *block_shape* is a tuple or list, the length of *block_shape* is M corresponding to the number of spatial dimensions. If *block_shape* is an int, the block size of M dimensions are the same, equal to *block_shape*. In this case of Ascend, M must be 2.

- **crops** (*Union[list(int), tuple(int)]*) −The crops values for spatial dimensions, containing M subtraction list. Each contains 2 integer values. All values must be >= 0. crops[i] specifies the crops values for spatial dimension i, which corresponds to input dimension i + offset,where offset = N-M, and N is the number of input dimensions. It is required that $input\_shape[i + offset] * block\_shape[i] > crops[i][0] + crops[i][1]$

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> block_shape = [2, 2]
>>> crops = [[0, 0], [0, 0]]
>>> input_x = mindspore.tensor([[[[1]]], [[[2]]], [[[3]]], [[[4]]]], mindspore.float32)
>>> output = mindspore.ops.batch_to_space_nd(input_x, block_shape, crops)
>>> print(output)
[[[[1.  2.]
   [3.  4.]]]]
```

mindspore.ops.bincount

`mindspore.ops.`**`bincount`**(*input*, *weights=None*, *minlength=0*)

Count the frequency of each value in the input tensor of non-negative ints.

If you don't specify *minlength*, the length of output tensor the length of the output tensor is max(*input*) + 1.

If *minlength* is specified, the length of the output tensor is max([max(*input*) + 1, *minlength*]).

If 'weights' is specified, the output results are weighted. If *n* is the value at position *i*, i.e `out[n] += weight[i]` instead of `out[n] += 1`.

Note: If *input* contains negative value, the result will be undefined.

Parameters

- **input** (`Tensor`) −1-D input tensor.
- **weights** (`Tensor, optional`) −Weights. Default `None` .
- **minlength** (*int, optional*) −A minimum number of bins for the output tensor. Default `0` .

Returns

Tensor, a tensor of shape [max(input)+1] if input is non-empty, otherwise, the shape is [0].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([2, 4, 1, 0, 0], dtype=mindspore.int64)
>>> print(mindspore.ops.bincount(x, minlength=7))
[2. 1. 1. 0. 1. 0. 0.]
>>> weights = mindspore.tensor([0, 0.25, 0.5, 0.75, 1], dtype=mindspore.float32)
>>> print(mindspore.ops.bincount(x, weights=weights))
[1.75 0.5  0.   0.   0.25]
```

mindspore.ops.block_diag

mindspore.ops.**block_diag**(**inputs*)

Creates a block diagonal matrix from the provided tensor.

Parameters

inputs (Tensor) – One or more tensors, the dimension of tensor should be 0, 1 or 2.

Returns

2-D Tensor, with all input tensors arranged in order so that their top left and bottom right corners are diagonally adjacent. All other elements are set to 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([[4], [3], [2]], mindspore.int32)
>>> x2 = mindspore.tensor([7, 6, 5], mindspore.int32)
>>> x3 = mindspore.tensor(1, mindspore.int32)
>>> x4 = mindspore.tensor([[5, 4, 3], [2, 1, 0]], mindspore.int32)
>>> x5 = mindspore.tensor([[8, 7], [7, 8]], mindspore.int32)
>>> out = mindspore.ops.block_diag(x1, x2, x3, x4, x5)
>>> print(out.asnumpy())
[[4 0 0 0 0 0 0 0 0 0]
 [3 0 0 0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0 0]
 [0 7 6 5 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 5 4 3 0 0]
 [0 0 0 0 0 2 1 0 0 0]
 [0 0 0 0 0 0 0 0 8 7]
 [0 0 0 0 0 0 0 0 7 8]]
```

mindspore.ops.broadcast_to

mindspore.ops.**broadcast_to**(*input*, *shape*)

Broadcasts input tensor to a given shape. The dim of input must be smaller than or equal to that of target. Suppose input shape is $(x_1, x_2, ..., x_m)$, target shape is $(*, y_1, y_2, ..., y_m)$, where $*$ means any additional dimension. The broadcast rules are as follows:

Compare the value of x_m and y_m, x_{m-1} and y_{m-1}, $\cdots$, x_1 and y_1 consecutively and decide whether these shapes are broadcastable and what the broadcast result is.

If the value pairs at a specific dim are equal, then that value goes right into that dim of output shape. With an input shape $(2, 3)$, target shape $(2, 3)$, the inferred output shape is $(2, 3)$.

If the value pairs are unequal, there are three cases:

Case 1: If the value of the target shape in the dimension is -1, the value of the output shape in the dimension is the value of the corresponding input shape in the dimension. With an input shape $(3, 3)$, target shape $(-1, 3)$, the output shape is $(3, 3)$.

Case 2: If the value of target shape in the dimension is not -1, but the corresponding value in the input shape is 1, then the corresponding value of the output shape is that of the target shape. With an input shape $(1, 3)$, target shape $(8, 3)$, the output shape is $(8, 3)$.

Case 3: If the corresponding values of the two shapes do not satisfy the above cases, it means that broadcasting from the input shape to the target shape is not supported.

So far we got the last m dims of the outshape, now focus on the first $*$ dims, there are two cases:

If the first $*$ dims of output shape does not have -1 in it, then fill the input shape with ones until their length are the same, and then refer to Case 2 mentioned above to calculate the output shape. With target shape $(3, 1, 4, 1, 5, 9)$, input shape $(1, 5, 9)$, the filled input shape will be $(1, 1, 1, 1, 5, 9)$ and thus the output shape is $(3, 1, 4, 1, 5, 9)$.

If the first $*$ dims of output shape have -1 in it, it implies this -1 is corresponding to a non-existing dim so they're not broadcastable. With target shape $(3, -1, 4, 1, 5, 9)$, input shape $(1, 5, 9)$, instead of operating the dim-filling process first, it raises errors directly.

Parameters

- **input** (Tensor) –The input tensor.
- **shape** (*tuple*) –The target shape.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> shape = (2, 3)
>>> x = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.broadcast_to(x, shape)
>>> print(output)
[[1. 2. 3.]
 [1. 2. 3.]]
>>> shape = (-1, 2)
>>> x = mindspore.tensor([[1], [2]], mindspore.float32)
>>> output = mindspore.ops.broadcast_to(x, shape)
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[1. 1.]
 [2. 2.]]
```

mindspore.ops.cat

mindspore.ops.**cat**(*tensors*, *axis=0*)

Connect input tensors along with the given axis.

Parameters

- **tensors** (*Union[tuple[Tensor], list[Tensor]]*) –The input tensors. The shapes of all axes except the specified concatenation *axis* should be equal.
- **axis** (*int*) –The specified axis. Default `0` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x1 = mindspore.tensor([[0, 1], [2, 1]], mindspore.float32)
>>> input_x2 = mindspore.tensor([[0, 1], [2, 1]], mindspore.float32)
>>> output = mindspore.ops.cat((input_x1, input_x2))
>>> print(output)
[[0. 1.]
 [2. 1.]
 [0. 1.]
 [2. 1.]]
>>> output = mindspore.ops.cat((input_x1, input_x2), 1)
>>> print(output)
[[0. 1. 0. 1.]
 [2. 1. 2. 1.]]
```

mindspore.ops.channel_shuffle

mindspore.ops.**channel_shuffle**(*x*, *groups*)

Divide the channels in a tensor of shape $(*, C, H, W)$ into g groups and rearrange them as $(*, \frac{C}{g}, g, H * W)$, while retaining the original tensor shape in the final output.

Parameters

- **x** (*Tensor*) –The input tensor.
- **groups** (*int*) –Number of groups to divide channels in.

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor(mindspore.ops.arange(0, 16, dtype=mindspore.int16).reshape(1, 4, 2,␣
↪2))
>>> y = mindspore.ops.channel_shuffle(x, groups=2)
>>> print(y)
[[[[ 0  1]
   [ 2  3]]
   [[ 8  9]
   [10 11]]
   [[ 4  5]
   [ 6  7]]
   [[12 13]
   [14 15]]]]
```

mindspore.ops.chunk

mindspore.ops.**chunk**(*input*, *chunks*, *axis=0*)

Split the input tensor into multiple sub-tensors along the specified axis.

Note: This function may return less than the specified number of chunks!

Parameters

- **input** (Tensor) −A tensor to be split.
- **chunks** (*int*) −The number of splits.
- **axis** (*int, optional*) −The axis along which to split. Default 0 .

Returns

Tuple of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.arange(0, 9, dtype=mindspore.float32)
>>> output = mindspore.ops.chunk(input, 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

mindspore.ops.column_stack

mindspore.ops.**column_stack**(*tensors*)

Creates a new tensor by horizontally stacking the tensors in *tensors*. Similar to *mindspore.ops.hstack()*.

Parameters

tensors (*Union[tuple[Tensor], list[Tensor]]*) – The tensors to be concatenated.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1, 1, 1])
>>> x2 = mindspore.tensor([2, 2, 2])
>>> output = mindspore.ops.column_stack((x1, x2))
>>> print(output)
[[1 2]
 [1 2]
 [1 2]]
```

mindspore.ops.concat

mindspore.ops.**concat**(*tensors*, *axis=0*)

Alias for *mindspore.ops.cat()*.

Tutorial Examples:

- Tensor - Tensor Operation
- Vision Transformer Image Classification - Building ViT as a whole
- Sentiment Classification Implemented by RNN - Dense

mindspore.ops.conj

mindspore.ops.**conj**(*input*)

Returns a tensor of complex numbers that are the complex conjugate of each element in input. The complex numbers in input must be of the form a + bj, where a is the real part and b is the imaginary part.

The complex conjugate returned by this operation is of the form a - bj.

If *input* is real, it is returned unchanged.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor(1.3+0.4j, mindspore.complex64)
>>> output = mindspore.ops.conj(x)
>>> output
Tensor(shape=[], dtype=Complex64, value= 1.3-0.4j)
```

mindspore.ops.count_nonzero

`mindspore.ops.`**`count_nonzero`**(*x*, *axis=()*, *keep_dims=False*, *dtype=mstype.int32*)

Counts the number of non-zero values in the input tensor along the given axis. If no axis is specified then all non-zeros in the tensor are counted.

Parameters

- **x** (`Tensor`) – The input tensor.
- **axis** (*`Union[int, tuple(int), list(int)], optional`*) – Specify the axis for computation. Default `()` , which counts all non-zero elements.
- **keep_dims** (*`bool, optional`*) – Whether to maintain dimensions specified by *axis*. Default `False` , don't keep these dimensions.
- **dtype** (*`Union[Number, mindspore.bool_], optional`*) – The data type returned. Default `mstype.int32` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: each value specified.
>>> x = mindspore.tensor([[0, 1, 0], [1, 1, 0]], mindspore.float32)
>>> nonzero_num = mindspore.ops.count_nonzero(x=x, axis=[0, 1], keep_dims=True,␣
↪dtype=mindspore.int32)
>>> print(nonzero_num)
[[3]]
>>> # case 2: all value is default.
>>> nonzero_num = mindspore.ops.count_nonzero(x=x)
>>> print(nonzero_num)
3
>>> # case 3: axis value was specified 0.
>>> nonzero_num = mindspore.ops.count_nonzero(x=x, axis=[0,])
>>> print(nonzero_num)
[1 2 0]
```

(continues on next page)

(continued from previous page)

```
>>> # case 4: axis value was specified 1.
>>> nonzero_num = mindspore.ops.count_nonzero(x=x, axis=[1,])
>>> print(nonzero_num)
[1 2]
>>> # case 5: keep_dims value was specified.
>>> nonzero_num = mindspore.ops.count_nonzero(x=x,  keep_dims=True)
>>> print(nonzero_num)
[[3]]
>>> # case 6: keep_dims and axis value was specified.
>>> nonzero_num = mindspore.ops.count_nonzero(x=x, axis=[0,], keep_dims=True)
>>> print(nonzero_num)
[[1 2 0]]
```

mindspore.ops.deepcopy

mindspore.ops.**deepcopy**(*input_x*)

Return a deepcopy of input tensor.

Parameters

input_x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[0, 1], [2, 1]], dtype=mindspore.int32)
>>> output = mindspore.ops.deepcopy(input)
>>> print(output)
[[0 1]
[2 1]]
```

mindspore.ops.diag

mindspore.ops.**diag**(*input*)

Return a tensor with the *input* as its diagonal elements and 0 elsewhere.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: When input is a 1-D tensor:
>>> input = mindspore.ops.randn(3)
>>> output = mindspore.ops.diag(input)
>>> print(output)
[[ 1.7477764  0.         0.       ]
 [ 0.        -1.2616369  0.       ]
 [ 0.         0.         2.3283238]]
>>>
>>> # case 2: When input is a multi-dimensional tensor:
>>> input = mindspore.ops.randn(2, 3)
>>> print(input)
[[ 0.21546374 -0.0120403  -0.7330481 ]
 [-2.5405762   0.44775972 -1.4063131 ]]
>>> output = mindspore.ops.diag(input)
>>> print(output)
[[[[ 0.21546374  0.          0.        ]
   [ 0.          0.          0.        ]]
  [[ 0.         -0.0120403   0.        ]
   [ 0.          0.          0.        ]]
  [[ 0.          0.         -0.7330481 ]
   [ 0.          0.          0.        ]]]
 [[[ 0.          0.          0.        ]
   [-2.5405762   0.          0.        ]]
  [[ 0.          0.          0.        ]
   [ 0.          0.44775972  0.        ]]
  [[ 0.          0.          0.        ]
   [ 0.          0.         -1.4063131 ]]]]
>>> # Assume input has dimensions (D_1,... D_k), the output is a tensor of rank 2k with␣
↪dimensions
    (D_1,..., D_k, D_1,..., D_k).
>>> print(output.shape)
(2, 3, 2, 3)
```

mindspore.ops.diagflat

mindspore.ops.**diagflat**(*input*, *offset=0*)

If *input* is a vector (1-D tensor), then returns a 2-D square tensor with the elements of *input* as the diagonal, If *input* is a tensor with more than one dimension, then returns a 2-D tensor with diagonal elements equal to a flattened *input*.

Parameters

- **input** (Tensor) –Input Tensor.
- **offset** (*int, optional*) –Diagonal offset. Default `0` .
  - When *offset* is a positive integer, shift the diagonal upward.
  - When *offset* is a negative integer, shift the diagonal downward.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> mindspore.ops.diagflat(mindspore.tensor([1, 2, 3]))
Tensor(shape=[3, 3], dtype=Int64, value=
[[1, 0, 0],
 [0, 2, 0],
 [0, 0, 3]])
>>> mindspore.ops.diagflat(mindspore.tensor([1, 2, 3]), 1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[0, 1, 0, 0],
 [0, 0, 2, 0],
 [0, 0, 0, 3],
 [0, 0, 0, 0]])
>>> mindspore.ops.diagflat(mindspore.tensor([[1, 2], [3, 4]]))
Tensor(shape=[4, 4], dtype=Int64, value=
[[1, 0, 0, 0],
 [0, 2, 0, 0],
 [0, 0, 3, 0],
 [0, 0, 0, 4]])
```

mindspore.ops.diagonal

`mindspore.ops.`**`diagonal`**(*input*, *offset=0*, *dim1=0*, *dim2=1*)

Returns diagonals of the input tensor along specified dimension.

If *input* is 2-D, returns a 1-D tensor containing the diagonal of *input* with the given offset.

If *input* has more than two dimensions, then the diagonals of specified 2-D sub-array determined by *dim1* and *dim2* is returned. The shape of returned tensor is the original shape with axis1 and axis2 removed and a new dimension inserted at the end corresponding to the diagonal.

Parameters

- **`input`** (`Tensor`) –The input tensor with at least two dimensions.
- **`offset`** (*`int, optional`*) –Diagonal offset. Default `0` .
  - When *offset* is a positive integer, shift the diagonal upward.
  - When *offset* is a negative integer, shift the diagonal downward.
- **`dim1`** (*`int, optional`*) –The first dimension specifying the 2D plane. Default `0` .
- **`dim2`** (*`int, optional`*) –The second dimension specifying the 2D plane. Default `1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[[1, 0, 0],
...                            [0, 2, 0],
...                            [0, 0, 3]],
...                           [[4, 0, 0],
...                            [0, 5, 0],
...                            [0, 0, 6]],
...                           [[7, 0, 0],
...                            [0, 8, 0],
...                            [0, 0, 9]]])
>>> mindspore.ops.diagonal(input)
Tensor(shape=[3, 3], dtype=Int64, value=
[[1, 0, 0],
 [0, 5, 0],
 [0, 0, 9]])
>>> mindspore.ops.diagonal(input, offset=1)
Tensor(shape=[3, 2], dtype=Int64, value=
[[0, 0],
 [2, 0],
 [0, 6]])
>>> mindspore.ops.diagonal(input, offset=0, dim1=2, dim2=1)
Tensor(shape=[3, 3], dtype=Int64, value=
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])
```

mindspore.ops.diagonal_scatter

mindspore.ops.**diagonal_scatter**(*input*, *src*, *offset=0*, *dim1=0*, *dim2=1*)

Embeds the values of the *src* tensor into *input* along the diagonal elements of input, with respect to *dim1* and *dim2* .

Note: Currently, `inf` value of elements in *input* or *src* is not supported.

Parameters

- **input** (Tensor) –The input tensor, whose dimension is larger than 1.
- **src** (Tensor) –The source tensor to embed.
- **offset** (*int, optional*) –Diagonal offset. Default `0` .
  - When *offset* is a positive integer, shift the diagonal upward.
  - When *offset* is a negative integer, shift the diagonal downward.
- **dim1** (*int, optional*) –First dimension with respect to which to take diagonal. Default `0` .
- **dim2** (*int, optional*) –Second dimension with respect to which to take diagonal. Default `1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.ops.zeros((3, 3))
>>> output = mindspore.ops.diagonal_scatter(input, mindspore.ops.ones(3), 0)
>>> print(output)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> output = mindspore.ops.diagonal_scatter(input, mindspore.ops.ones(2), 1)
>>> print(output)
[[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

mindspore.ops.diag_embed

mindspore.ops.**diag_embed**(*input*, *offset=0*, *dim1=- 2*, *dim2=- 1*)

Create a tensor whose diagonals of certain 2D planes (specified by *dim1* and *dim2*) are filled by *input*, and all other positions are set to `0`. The 2D planes formed by the last two dimensions of the returned tensor are chosen by default.

Parameters

- **input** (Tensor) −Values to fill diagonal.
- **offset** (*int, optional*) −Diagonal offset. Default `0` .
  - When *offset* is a positive integer, shift the diagonal upward.
  - When *offset* is a negative integer, shift the diagonal downward.
- **dim1** (*int, optional*) −The first dimension for diagonal filling. Default `-2` .
- **dim2** (*int, optional*) −The second dimension for diagonal filling. Default `-1` .

Returns

A tensor with the same dtype as *input*, and with shape that has one dimension higher than the *input*.

Raises

ValueError −If the dimension of *input* is not 1D-6D.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3],
...                           [4, 5, 6],
...                           [7, 8, 9]])
>>> mindspore.ops.diag_embed(input)
Tensor(shape=[3, 3, 3], dtype=Int64, value=
[[[1, 0, 0], [0, 2, 0], [0, 0, 3]],
 [[4, 0, 0], [0, 5, 0], [0, 0, 6]],
 [[7, 0, 0], [0, 8, 0], [0, 0, 9]]])
```

(continues on next page)

(continued from previous page)

```
>>> mindspore.ops.diag_embed(input, offset=1, dim1=0, dim2=1)
Tensor(shape=[4, 4, 3], dtype=Int64, value=
[[[0, 0, 0], [1, 4, 7], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [2, 5, 8], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0], [3, 6, 9]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]])
```

mindspore.ops.dyn_shape

mindspore.ops.**dyn_shape**(*input_x*)

Returns the shape of the input tensor.

Parameters

input_x (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor(mindspore.ops.ones(shape=[3, 2, 1]), mindspore.float32)
>>> output = mindspore.ops.dyn_shape(input_x)
>>> print(output)
[3 2 1]
```

mindspore.ops.dsplit

mindspore.ops.**dsplit**(*input*, *indices_or_sections*)

Splits a tensor along the 3rd axis. It is equivalent to *ops.tensor_split* with $axis = 2$.

Parameters

- **input** (Tensor) –A tensor to be divided.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –See *indices_or_sections* argument in *mindspore.ops.tensor_split()*.

Returns

Tuple of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.arange(16.0).reshape(2, 2, 4)
>>> print(input)
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]]
 [[ 8.  9. 10. 11.]
  [12. 13. 14. 15.]]]
>>> output = mindspore.ops.dsplit(input, 2)
>>> print(output)
(Tensor(shape=[2, 2, 2], dtype=Float32, value=
[[[ 0.00000000e+00,  1.00000000e+00],
  [ 4.00000000e+00,  5.00000000e+00]],
 [[ 8.00000000e+00,  9.00000000e+00],
  [ 1.20000000e+01,  1.30000000e+01]]]), Tensor(shape=[2, 2, 2], dtype=Float32, value=
[[[ 2.00000000e+00,  3.00000000e+00],
  [ 6.00000000e+00,  7.00000000e+00]],
 [[ 1.00000000e+01,  1.10000000e+01],
  [ 1.40000000e+01,  1.50000000e+01]]]))
>>> output = mindspore.ops.dsplit(input, [3, 6])
>>> print(output)
(Tensor(shape=[2, 2, 3], dtype=Float32, value=
[[[ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00],
  [ 4.00000000e+00,  5.00000000e+00,  6.00000000e+00]],
 [[ 8.00000000e+00,  9.00000000e+00,  1.00000000e+01],
  [ 1.20000000e+01,  1.30000000e+01,  1.40000000e+01]]]), Tensor(shape=[2, 2, 1],␣
↪dtype=Float32, value=
[[[ 3.00000000e+00],
  [ 7.00000000e+00]],
 [[ 1.10000000e+01],
  [ 1.50000000e+01]]]), Tensor(shape=[2, 2, 0], dtype=Float32, value=
))
```

mindspore.ops.dstack

mindspore.ops.**dstack**(*tensors*)

Stacks tensors along the third axis.

Note:

- 1-D tensors $(N,)$ should be reshaped to $(1, N, 1)$. 2-D tensors (M, N) should be reshaped to $(M, N, 1)$ before concatenation.
- The tensors must have the same shape along all but the third axis. 1-D or 2-D tensors must have the same shape.

Parameters

tensors (*Union(List[Tensor], Tuple[Tensor])*) – The list of tensors or tuple of tensors.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor(mindspore.ops.arange(1, 7).reshape(2, 3))
>>> x2 = mindspore.tensor(mindspore.ops.arange(7, 13).reshape(2, 3))
>>> out = mindspore.ops.dstack([x1, x2])
>>> print(out.asnumpy())
[[[ 1.  7.]
  [ 2.  8.]
  [ 3.  9.]]
 [[ 4. 10.]
  [ 5. 11.]
  [ 6. 12.]]]
```

mindspore.ops.einsum

mindspore.ops.**einsum**(*equation*, **operands*)

According to the Einstein summation Convention (Einsum), the product of the input tensor elements is summed along the specified dimension.

Note:

- The sublist format is also supported. For example, ops.einsum(op1, sublist1, op2, sublist2, ···, sublist_out). In this format, equation can be derived by the sublists which are made up of Python's Ellipsis and list of integers in [0, 52). Each operand is followed by a sublist and an output sublist is at the end.
- The value can contain only letters, commas, ellipsis and arrow. The letters represent input tensor dimension, commas represent separate tensors, ellipsis indicates the tensor dimension that you do not care about, the left of the arrow indicates the input tensors, and the right of it indicates the desired output dimension.

Parameters

- **equation** (*str*) –Notation based on the Einstein summation convention.
- **operands** (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

GPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> equation = "i->"
>>> output = mindspore.ops.einsum(equation, x)
>>> print(output)
[7.]
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> y = mindspore.tensor([2.0, 4.0, 3.0], mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> equation = "i,i->i"
>>> output = mindspore.ops.einsum(equation, x, y)
>>> print(output)
[ 2. 8. 12.]
>>> x = mindspore.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], mindspore.float32)
>>> y = mindspore.tensor([[2.0, 3.0], [1.0, 2.0], [4.0, 5.0]], mindspore.float32)
>>> equation = "ij,jk->ik"
>>> output = mindspore.ops.einsum(equation, x, y)
>>> print(output)
[[16. 22.]
 [37. 52.]]
>>> x = mindspore.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], mindspore.float32)
>>> equation = "ij->ji"
>>> output = mindspore.ops.einsum(equation, x)
>>> print(output)
[[1. 4.]
 [2. 5.]
 [3. 6.]]
>>> x = mindspore.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], mindspore.float32)
>>> equation = "ij->j"
>>> output = mindspore.ops.einsum(equation, x)
>>> print(output)
[5. 7. 9.]
>>> x = mindspore.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]], mindspore.float32)
>>> equation = "...->"
>>> output = mindspore.ops.einsum(equation, x)
>>> print(output)
[21.]
>>> x = mindspore.tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> y = mindspore.tensor([2.0, 4.0, 1.0], mindspore.float32)
>>> equation = "j,i->ji"
>>> output = mindspore.ops.einsum(equation, x, y)
>>> print(output)
[[ 2. 4. 1.]
 [ 4. 8. 2.]
 [ 6. 12. 3.]]
>>> x = mindspore.tensor([1, 2, 3, 4], mindspore.float32)
>>> y = mindspore.tensor([1, 2], mindspore.float32)
>>> output = mindspore.ops.einsum(x, [..., 1], y, [..., 2], [..., 1, 2])
>>> print(output)
[[1. 2.]
 [2. 4.]
 [3. 6.]
 [4. 8.]]
```

mindspore.ops.expand_dims

mindspore.ops.**expand_dims**(*input_x*, *axis*)

Adds an additional axis to input tensor.

Note:

- The dimension of *input_x* should be greater than or equal to 1.
- If the specified axis is a negative number, the index is counted backward from the end and starts at 1.

Parameters

- **input_x** (Tensor) –The input tensor.
- **axis** (*int*) –The newly added axis. Only constant value is allowed.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_tensor = mindspore.tensor([[2, 2], [2, 2]], mindspore.float32)
>>> output = mindspore.ops.expand_dims(input_tensor, 0)
>>> print(output)
[[[2. 2.]
  [2. 2.]]]
```

mindspore.ops.flip

mindspore.ops.**flip**(*input*, *dims*)

Reverses elements in a tensor along the given dims.

Parameters

- **input** (Tensor) –The input tensor.
- **dims** (*Union[list[int], tuple[int]]*) –The dimension to flip.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.arange(1, 9).reshape((2, 2, 2)))
>>> output = mindspore.ops.flip(input, (0, 2))
>>> print(output)
[[[6 5]
  [8 7]]
 [[2 1]
  [4 3]]]
```

mindspore.ops.fliplr

mindspore.ops.**fliplr**(*input*)

Flip the input tensor in left/right direction.

Parameters

input (Tensor) −The input tensor, the dimension must be at least 2.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.arange(1, 9).reshape((2, 2, 2)))
>>> output = mindspore.ops.fliplr(input)
>>> print(output)
[[[3 4]
  [1 2]]
 [[7 8]
  [5 6]]]
```

mindspore.ops.flipud

mindspore.ops.**flipud**(*input*)

Flip the input tensor in up/down direction.

Parameters

input (Tensor) −The input tensor, the dimension must be at least 2.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

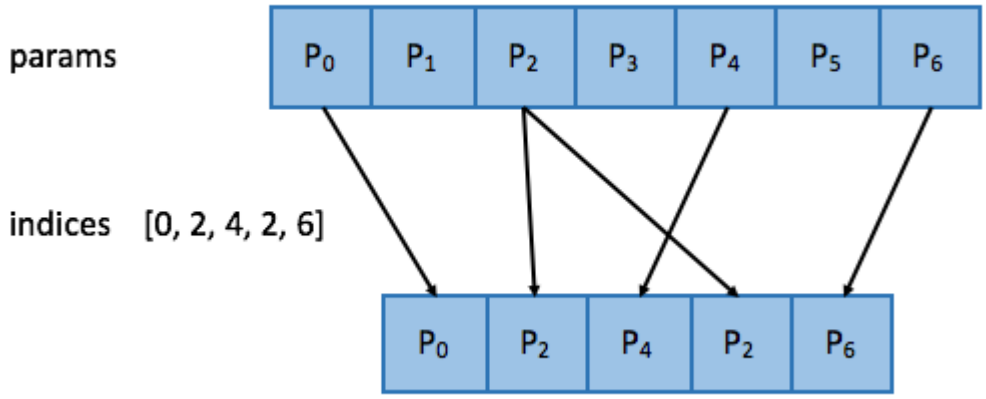
```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.arange(1, 9).reshape((2, 2, 2)))
>>> output = mindspore.ops.flipud(input)
>>> print(output)
[[[5 6]
  [7 8]]
 [[1 2]
  [3 4]]]
```

mindspore.ops.gather

mindspore.ops.**gather**(*input_params*, *input_indices*, *axis*, *batch_dims=0*)

Returns the slice of the input tensor corresponding to the elements of *input_indices* on the specified *axis*.

The following figure shows the calculation process of Gather commonly:



where params represents the input *input_params*, and indices represents the index to be sliced *input_indices*.

Note:

- The value of input_indices must be in the range of *[0, input_param.shape[axis])*. On CPU and GPU, an error is raised if an out of bound indice is found. On Ascend, the results may be undefined.
- The data type of input_params cannot be *mindspore.bool_* .
- The shape of returned tensor is $input\_params.shape[:axis] + input\_indices.shape[batch\_dims:] + input\_params.shape[axis+1:]$.

Parameters

- **input_params** (Tensor) –The input Tensor.
- **input_indices** (Tensor) –The specified indices.
- **axis** (*Union(int, Tensor[int])*) –The specified axis.
- **batch_dims** (*int*) –The number of batch dimensions. Default 0 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case1: input_indices is a Tensor with shape (5, ).
>>> input_params = mindspore.tensor([1, 2, 3, 4, 5, 6, 7], mindspore.float32)
>>> input_indices = mindspore.tensor([0, 2, 4, 2, 6], mindspore.int32)
>>> axis = 0
>>> output = mindspore.ops.gather(input_params, input_indices, axis)
>>> print(output)
[1. 3. 5. 3. 7.]
>>> # case2: input_indices is a Tensor with shape (2, 2). When the input_params has one␣
 ↪dimension,
>>> # the output shape is equal to the input_indices shape.
>>> input_indices = mindspore.tensor([[0, 2], [2, 6]], mindspore.int32)
>>> axis = 0
>>> output = mindspore.ops.gather(input_params, input_indices, axis)
>>> print(output)
[[1. 3.]
 [3. 7.]]
>>> # case3: input_indices is a Tensor with shape (2, ) and
>>> # input_params is a Tensor with shape (3, 4) and axis is 0.
>>> input_params = mindspore.tensor([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], mindspore.
 ↪float32)
>>> input_indices = mindspore.tensor([0, 2], mindspore.int32)
>>> axis = 0
>>> output = mindspore.ops.gather(input_params, input_indices, axis)
>>> print(output)
[[ 1.  2.  3.  4.]
 [ 9. 10. 11. 12.]]
>>> # case4: input_indices is a Tensor with shape (2, ) and
>>> # input_params is a Tensor with shape (3, 4) and axis is 1, batch_dims is 1.
>>> input_params = mindspore.tensor([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], mindspore.
 ↪float32)
>>> input_indices = mindspore.tensor([0, 2, 1], mindspore.int32)
>>> axis = 1
>>> batch_dims = 1
>>> output = mindspore.ops.gather(input_params, input_indices, axis, batch_dims)
>>> print(output)
[ 1.  7. 10.]
```

mindspore.ops.gather_d

mindspore.ops.**gather_d**(*x*, *dim*, *index*)

Gathers elements along an axis specified by dim.

Refer to *mindspore.ops.gather_elements()* for more detail.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2], [3, 4]], mindspore.int32)
>>> index = mindspore.tensor([[0, 0], [1, 0]], mindspore.int32)
>>> dim = 1
>>> output = mindspore.ops.gather_d(x, dim, index)
>>> print(output)
[[1 1]
 [4 3]]
```

mindspore.ops.gather_elements

mindspore.ops.**gather_elements**(*input*, *dim*, *index*)

Gathers elements along the specified dim and indices.

Note: *input* and *index* have the same length of dimensions, and *index.shape[axis]* <= *input.shape[axis]* where axis goes through all dimensions of *input* except *dim*.

Warning: On Ascend, the behavior is unpredictable in the following cases:

- the value of *index* is not in the range *[-input.shape[dim], input.shape[dim])* in forward;
- the value of *index* is not in the range *[0, input.shape[dim])* in backward.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*int*) –The specified dim.
- **index** (Tensor) –The specified indices.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2], [3, 4]], mindspore.int32)
>>> index = mindspore.tensor([[0, 0], [1, 0]], mindspore.int32)
>>> dim = 1
>>> output = mindspore.ops.gather_elements(x, dim, index)
>>> print(output)
[[1 1]
 [4 3]]
```

mindspore.ops.gather_nd

mindspore.ops.**gather_nd**(*input_x*, *indices*)

Gathers slices from the input tensor by specified indices.

Suppose *indices* is an K-dimensional integer tensor, follow the formula below:

$$output[(i_0, ..., i_{K-2})] = input\_x[indices[(i_0, ..., i_{K-2})]]$$

Must be satisfied $indices.shape[-1] <= input\_x.rank$.

Parameters

- **input_x** (Tensor) –The input tensor.
- **indices** (Tensor) –The specified indices.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> input_x = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> indices = mindspore.tensor([[0, 0], [1, 1]], mindspore.int32)
>>> output = mindspore.ops.gather_nd(input_x, indices)
>>> print(output)
[-0.1  0.5]
```

mindspore.ops.hstack

mindspore.ops.**hstack**(*tensors*)

Stacks tensors in sequence horizontally.

Note:

- Dynamic rank input of 8-D tensors with type float64 is not supported in graph mode (mode=mindspore.GRAPH_MODE).
- This is equivalent to concatenation along the second axis, except for 1-D tensors where it concatenates along the first axis.
- The tensors must have the same shape along all but the second axis, except 1-D tensors which can be any length.

Parameters

tensors (*Union[tuple[Tensor], list[Tensor]]*) –Tuple of tensors or list of tensors.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1, 1, 1])
>>> x2 = mindspore.tensor([2, 2, 2])
>>> output = mindspore.ops.hstack((x1, x2))
>>> print(output)
[1. 1. 1. 2. 2. 2.]
```

mindspore.ops.hsplit

mindspore.ops.**hsplit**(*input*, *indices_or_sections*)

Splits a tensor into multiple sub-tensors horizontally. It is equivalent to *ops.tensor_split* with $axis = 1$.

Parameters

- **input** (Tensor) –The input tensor.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –See argument in *mindspore.ops.tensor_split()*.

Returns

Tuple of tensors

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.ops.arange(0, 6).reshape((2, 3))
>>> output = mindspore.ops.hsplit(mindspore.tensor(input_x, mindspore.float32), 3)
>>> print(output)
(Tensor(shape=[2, 1], dtype=Float32, value=[[ 0.00000000e+00], [ 3.00000000e+00]]),
 Tensor(shape=[2, 1], dtype=Float32, value=[[ 1.00000000e+00], [ 4.00000000e+00]]),
 Tensor(shape=[2, 1], dtype=Float32, value=[[ 2.00000000e+00], [ 5.00000000e+00]]))
```

mindspore.ops.index_add

mindspore.ops.**index_add**(*x*, *indices*, *y*, *axis*, *use_lock=True*, *check_index_bound=True*)

Add the elements of input *y* into input *x* along the given axis and indices.

Note:

- *indices* is a one-dimensional tensor, and $indices.shape[0] = y.shape[axis]$.
- The value range of the elements in *indices* is $[0, x.shape[axis] - 1]$.

Parameters

- **x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.

- **y** (Tensor) −The input tensor to add to *x*.
- **axis** (*int*) −The specified axis.
- **use_lock** (*bool, optional*) −Whether to enable a lock to protect the updating process of variable tensors. Default `True` .
- **check_index_bound** (*bool, optional*) −Whether to check index boundary. Default `True` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.Parameter(mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.
↪float32),
...                         name="name_x")
>>> indices = mindspore.tensor([0, 2], mindspore.int32)
>>> y = mindspore.tensor([[0.5, 1.0], [1.0, 1.5], [2.0, 2.5]], mindspore.float32)
>>> output = mindspore.ops.index_add(x, indices, y, 1)
>>> print(output)
[[ 1.5  2.   4. ]
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
```

mindspore.ops.index_fill

mindspore.ops.**index_fill**(*x*, *axis*, *index*, *value*)

Fills the elements of the input *x* with *value* along the given axis and indices.

Parameters

- **x** (Tensor) −The input tensor.
- **axis** (*Union[int, Tensor]*) −The specified axis.
- **index** (Tensor) −The specified indices.
- **value** (*Union[bool, int, float, Tensor]*) −Value to fill the returned tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.float32)
>>> index = mindspore.tensor([0, 2], mindspore.int32)
>>> value = mindspore.tensor(-2.0, mindspore.float32)
>>> y = mindspore.ops.index_fill(x, 1, index, value)
>>> print(y)
[[-2. 2. -2.]
 [-2. 5. -2.]
 [-2. 8. -2.]]
```

mindspore.ops.index_select

mindspore.ops.**index_select**(*input*, *axis*, *index*)

Select the input tensor according to the specified axis and index and return a new tensor.

Note:

- The value of *index* must be in the range of *[0, input.shape[axis])*, the result is undefined out of range.
- The returned tensor has the same number of dimensions as the input tensor.The *axis* dimension has the same size as the length of *index* , other dimensions have the same size as the input tensor.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*int*) –The specified axis.
- **index** (Tensor) –The specified indices, a 1-D tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.arange(0, 16).reshape(2, 2, 4), mindspore.float32)
>>> print(input)
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]]
 [[ 8.  9. 10. 11.]
  [12. 13. 14. 15.]]]
>>> index = mindspore.tensor([0,], mindspore.int32)
>>> y = mindspore.ops.index_select(input, 1, index)
>>> print(y)
[[[ 0.  1.  2.  3.]]
 [[ 8.  9. 10. 11.]]]
```

mindspore.ops.inplace_add

mindspore.ops.**inplace_add**(*x*, *v*, *indices*)

Add *x* to *v* according to the indices.

for each *i*, ⋯, *j* in *indices* :

$$x[\text{indices}[i, ..., j]] \mathrel{+}= y[i, ..., j]$$

Parameters

- **x** (Tensor) −The input tensor.
- **v** (Tensor) −The input tensor to add to *x*.
- **indices** (*Union[int, tuple]*) −Indices into the input *x* along the 0th dimension.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> indices = (0, 1)
>>> x = mindspore.tensor([[1, 2], [3, 4], [5, 6]], mindspore.float32)
>>> input_v = mindspore.tensor([[0.5, 1.0], [1.0, 1.5]], mindspore.float32)
>>> output = mindspore.ops.inplace_add(x, input_v, indices)
>>> print(output)
[[1.5 3. ]
 [4.  5.5]
 [5.  6. ]]
```

mindspore.ops.inplace_index_add

mindspore.ops.**inplace_index_add**(*var*, *indices*, *updates*, *axis*)

Add *updates* to *var* according to the indices and axis.

for each *i*, ⋯, *j* in *indices* :

$$x[:, \text{indices}[i, ..., j], :] \mathrel{+}= v[:, i, ..., j, :]$$

where *i* is the index of the element in *indices*, and the axis of *indices[i]* is determined by the input *axis*.

Parameters

- **var** (*Union[Parameter, Tensor]*) −The input parameter or tensor.
- **indices** (Tensor) −The specified indices, a 1-D tensor.
- **updates** (Tensor) −The input tensor to add to *var*.
- **axis** (*int*) −The specified axis.

Returns

Tensor

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> var = mindspore.Parameter(mindspore.tensor([[1, 2], [3, 4], [5, 6]], mindspore.float32))
>>> indices = mindspore.tensor([0, 1], mindspore.int32)
>>> updates = mindspore.tensor([[0.5, 1.0], [1.0, 1.5]], mindspore.float32)
>>> mindspore.ops.inplace_index_add(var, indices, updates, axis=0)
>>> print(var.asnumpy())
[[1.5 3. ]
 [4.  5.5]
 [5.  6. ]]
```

mindspore.ops.inplace_sub

`mindspore.ops.`**`inplace_sub`**(*x*, *v*, *indices*)

Subtract *v* in *x* according to the indices.

for each *i*, ···, *j* in *indices* :

$$x[\text{indices}[i, ..., j]] \mathrel{-}= v[i, ..., j]$$

Parameters

- **x** (`Tensor`) −The input tensor.
- **v** (`Tensor`) −The input tensor to subtract from *x* .
- **indices** (*`Union[int, tuple]`*) −Indices into the input *x* along the 0th dimension.

Returns

Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> indices = (0, 1)
>>> x = mindspore.tensor([[1, 2], [3, 4], [5, 6]], mindspore.float32)
>>> input_v = mindspore.tensor([[0.5, 1.0], [1.0, 1.5]], mindspore.float32)
>>> output = mindspore.ops.inplace_sub(x, input_v, indices)
>>> print(output)
[[0.5 1. ]
 [2.  2.5]
 [5.  6. ]]
```

mindspore.ops.inplace_update

mindspore.ops.**inplace_update**(*x*, *v*, *indices*)

Updates *x* to *v* according to the indices.

> **Warning:** This is an experimental API that is subject to change or deletion.

for each *i*, ···, *j* in *indices* :

$$x[extindices[i, ..., j]] = v[i, ..., j]$$

Parameters

- **x** (Tensor) −The input tensor.
- **v** (Tensor) −The input tensor to update to *x*.
- **indices** (*Union[int, tuple[int], Tensor]*) −Indices into the input *x* along the 0th dimension.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> indices = (0, 1)
>>> x = mindspore.tensor([[1, 2], [3, 4], [5, 6]], mindspore.float32)
>>> v = mindspore.tensor([[0.5, 1.0], [1.0, 1.5]], mindspore.float32)
>>> output = mindspore.ops.inplace_update(x, v, indices)
>>> print(output)
[[0.5 1. ]
 [1.  1.5]
 [5.  6. ]]
```

mindspore.ops.is_nonzero

mindspore.ops.**is_nonzero**(*input*)

Determine whether the input Tensor contains `0` or `False`. The input can only be a single element.

Parameters

input (Tensor) −The input tensor.

Returns

Bool

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([[[False]]])
>>> x2 = mindspore.tensor([[3.5]])
>>> out1 = mindspore.ops.is_nonzero(x1)
>>> print(out1)
False
>>> out2 = mindspore.ops.is_nonzero(x2)
>>> print(out2)
True
```

mindspore.ops.masked_fill

mindspore.ops.**masked_fill**(*input_x*, *mask*, *value*)

Fills elements of tensor with value where mask is `True`.

Support broadcast.

Parameters

- **input_x** (Tensor) –The input tensor.
- **mask** (Tensor[*bool*]) –The input mask.
- **value** (*Union[Number, Tensor]*) –The value to fill in with.

Returns

Tensor

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([1., 2., 3., 4.], mindspore.float32)
>>> mask = mindspore.tensor([True, True, False, True], mindspore.bool_)
>>> output = mindspore.ops.masked_fill(input_x, mask, 0.5)
>>> print(output)
[0.5 0.5 3.  0.5]
```

mindspore.ops.masked_select

mindspore.ops.**masked_select**(*input*, *mask*)

Return a new 1-D tensor which indexes the *input* tensor according to the boolean *mask*.

Support broadcast.

Parameters

- **input** (Tensor) –The input tensor.
- **mask** (Tensor[*bool*]) –The input mask.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3, 4], mindspore.int64)
>>> mask = mindspore.tensor([1, 0, 1, 0], mindspore.bool_)
>>> output = mindspore.ops.masked_select(x, mask)
>>> print(output)
[1 3]
```

mindspore.ops.meshgrid

`mindspore.ops.`**`meshgrid`**(**inputs*, *indexing='xy'*)

Creates grids of coordinates specified by the 1D inputs。

Note:

- In graph mode, a tuple of N 1-D tensors and N should be greater than 1.
- In pynative mode, a tuple of N 0-D or 1-D tensors and N should be greater than 0. The data type is Number.
- In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for `'xy'` indexing and (M, N) for `'ij'` indexing.
- In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for `'xy'` indexing and (M, N, P) for `'ij'` indexing.

Parameters

`inputs` (*`Union[tuple[Tensor], list[Tensor]]`*) – Tuple of tensors or list of tensors.

Keyword Arguments

`indexing` (*`str, optional`*) – Cartesian ('xy', default) or matrix ('ij') indexing of output. Valid options `'xy'` or `'ij'`. Default `'xy'` .

Returns

Tuple of N N-D tensors

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3, 4], mindspore.int32)
>>> y = mindspore.tensor([5, 6, 7], mindspore.int32)
>>> z = mindspore.tensor([8, 9, 0, 1, 2], mindspore.int32)
>>> output = mindspore.ops.meshgrid(x, y, z, indexing='xy')
>>> print(output)
(Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]]]),
 Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5]],
  [[6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6]],
  [[7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7]]]),
 Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]]]))
```

mindspore.ops.narrow

mindspore.ops.**narrow**(*input*, *axis*, *start*, *length*)

Slice the tensor from the *start* position with a length of *length* along *axis* .

Parameters

- **input** (Tensor) – The input tensor.
- **axis** (*int*) – the specified axis.
- **start** (*int*) – the specified starting position.
- **length** (*int*) – the specified length.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.int32)
>>> output = mindspore.ops.narrow(x, 0, 0, 2)
>>> print(output)
[[ 1 2 3]
 [ 4 5 6]]
>>> output = mindspore.ops.narrow(x, 1, 1, 2)
>>> print(output)
[[ 2 3]
 [ 5 6]
 [ 8 9]]
```

mindspore.ops.moveaxis

mindspore.ops.**moveaxis**(*x*, *source*, *destination*)

Move axis of an array from source to destination.

Refer to *mindspore.ops.movedim()* for more detail.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.ops.zeros((3, 4, 5))
>>> output = mindspore.ops.moveaxis(x, 0, -1)
>>> print(output.shape)
(4, 5, 3)
```

mindspore.ops.movedim

mindspore.ops.**movedim**(*x*, *source*, *destination*)

Swap two dimensions of the input tensor.

Parameters

- **x** (Tensor) –The input tensor.
- **source** (*Union[int, sequence[int]]*) –Original dimensions.
- **destination** (*Union[int, sequence[int]]*) –Destination positions for each of the original dimensions.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> # case1 : moving single axis
>>> import mindspore
>>> x = mindspore.tensor(mindspore.ops.zeros((3, 4, 5)))
>>> output = mindspore.ops.movedim(x, 0, -1)
>>> print(output.shape)
(4, 5, 3)
>>> # case 2 : moving multiple axes
>>> x = mindspore.tensor(mindspore.ops.zeros((3, 4, 5)))
>>> output = mindspore.ops.movedim(x, (0, 2), (1, 2))
>>> print(output.shape)
(4, 3, 5)
```

mindspore.ops.nan_to_num

mindspore.ops.**nan_to_num**(*input*, *nan=None*, *posinf=None*, *neginf=None*)

Replace the *NaN*, positive infinity and negative infinity values in *input* with the specified values in *nan*, *posinf* and *neginf* respectively.

> **Warning:** For Ascend, it is only supported on Atlas A2 Training Series Products. This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **nan** (*number, optional*) –The replace value of *NaN*. Default None.
- **posinf** (*number, optional*) –the value to replace *posinf* values with. Default None, replacing *posinf* with the maximum value supported by the data type of *input*.
- **neginf** (*number, optional*) –the value to replace *neginf* values with. Default None, replacing *neginf* with the minimum value supported by the data type of *input*.

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([float('nan'), float('inf'), -float('inf'), 5.0], mindspore.
↪float32)
>>> output = mindspore.ops.nan_to_num(input)
>>> print(output)
[ 0.0000000e+00  3.4028235e+38 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.ops.nan_to_num(input, 1.0)
>>> print(output)
[ 1.0000000e+00  3.4028235e+38 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.ops.nan_to_num(input, 1.0, 2.0)
>>> print(output)
[ 1.0000000e+00  2.0000000e+00 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.ops.nan_to_num(input, 1.0, 2.0, 3.0)
>>> print(output)
[1.  2.  3.  5.0]
```

mindspore.ops.nanmean

mindspore.ops.**nanmean**(*input*, *axis=None*, *keepdims=False*, *, *dtype=None*)

Computes the mean of *input* in specified dimension, ignoring NaN.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*int, optional*) –The specified axis for computation. Default None .
- **keepdims** (*bool, optional*) –Whether the output tensor has to dim retained. Default False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[0.5, -1.1, float('nan')], [3.4, float('nan'), float('nan')]],␣
↪mindspore.float32)
>>> y = mindspore.ops.nanmean(x, axis=0, keepdims=False)
>>> print(y)
[ 1.95 -1.1    nan]
```

mindspore.ops.nanmedian

mindspore.ops.**nanmedian**(*input*, *axis=- 1*, *keepdims=False*)

Computes the median and indices of *input* in specified dimension, ignoring NaN.

> **Warning:** *indices* does not necessarily contain the first occurrence of each median value found in the *input*, unless it is unique.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*int, optional*) –The specified axis for computation. Default `-1` .
- **keepdims** (*bool, optional*) –Whether the output tensor needs to retain dimension or not. Default `False` .

Returns

Tuple(median, median_indices)

Supported Platforms:

CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[0.57, 0.11, float("nan")],
...                       [0.38, float("nan"), float("nan")],
...                       [0.36, 0.16, float("nan")]], mindspore.float32)
>>> y, idx = mindspore.ops.nanmedian(x, axis=0, keepdims=False)
>>> print(y)
[0.38 0.11  nan]
>>> print(idx)
[1 0 0]
```

mindspore.ops.nansum

mindspore.ops.**nansum**(*input*, *axis=None*, *keepdims=False*, *, *dtype=None*)

Computes sum of *input* over a given dimension, ignoring NaN.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*Union[int, tuple(int)], optional*) –The dimensions to reduce. Supposed the rank of *input* is r, axis must be in the range [-rank(input), rank(input)). Default None, all dimensions are reduced.
- **keepdims** (*bool, optional*) –Whether the output tensor keeps has dim retained. Default False.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The dtype of output tensor. Default None.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[float("nan"), 2, 3], [1, float("nan"), 3], [1, 2, float("nan")]],␣
↪mindspore.float32)
>>> # case1: axis is None, keepdims is False,
>>> output1 = mindspore.ops.nansum(x, axis=None, dtype=mindspore.float32)
>>> print(output1)
12.0
>>> # case2: axis is int, set as 0, and keepdims is False
>>> output2 = mindspore.ops.nansum(x, axis=0, dtype=mindspore.float32)
>>> print(output2)
[2. 4. 6.]
>>> # case3: axis is int, set as 0, and keepdims is False
>>> output3 = mindspore.ops.nansum(x, axis=0, keepdims=True, dtype=mindspore.float32)
>>> print(output3)
[[2. 4. 6.]]
>>> # case4: axis is tuple(int) or list(int), set as (0, 1), and keepdims is False
>>> output4 = mindspore.ops.nansum(x, axis=(0, 1), dtype=mindspore.float32)
>>> print(output4)
12.0
```

mindspore.ops.normal

mindspore.ops.**normal**(*shape*, *mean*, *stddev*, *seed=None*)

Return a random tensor that conforms to the normal (Gaussian) distribution.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **shape** (*tuple*) –The shape of returned tensor.
- **mean** (*Union[Tensor, int, float]*) –The mean of the normal distribution for the returned tensor.
- **stddev** (*Union[Tensor, int, float]*) –The standard deviation of the normal distribution for the returned tensor.
- **seed** (*int, optional*) –Random seed. Default: None , which is equivalent to 0.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> shape = (3, 1, 2)
>>> mean = mindspore.tensor([[3, 4], [5, 6]], mindspore.float32)
>>> stddev = mindspore.tensor(1.0, mindspore.float32)
>>> output = mindspore.ops.normal(shape, mean, stddev, seed=5)
>>> result = output.shape
>>> print(result)
(3, 2, 2)
>>> shape = (3, 1, 3)
>>> mean = mindspore.tensor([[3, 4, 3], [3, 5, 6]], mindspore.float32)
>>> stddev = mindspore.tensor(1.0, mindspore.float32)
>>> output = mindspore.ops.normal(shape, mean, stddev, seed=5)
>>> result = output.shape
>>> print(result)
(3, 2, 3)
>>> shape = (3, 1, 3)
>>> mean = mindspore.tensor([[1, 2, 3], [3, 4, 3], [3, 5, 6]], mindspore.float32)
>>> stddev = mindspore.tensor(1.0, mindspore.float32)
>>> output = mindspore.ops.normal(shape, mean, stddev, seed=5)
>>> result = output.shape
>>> print(result)
(3, 3, 3)
```

mindspore.ops.nonzero

mindspore.ops.**nonzero**(*input*, *, *as_tuple=False*)

Return the positions of all non-zero values.

Parameters

input (Tensor) – The input tensor.

Note:

- Ascend: Rank of Input tensor can be equal to 0 except jit level O2 mode.
- CPU/GPU: Rank of Input tensor should be greater than or eaqual to 1.

Keyword Arguments

as_tuple (bool, *optional*) – Whether the output is tuple. Default False .

Returns

Tensor or tuple of tensors

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[[1,  0], [-5, 0]]], mindspore.int32)
>>> output = mindspore.ops.nonzero(x)
>>> print(output)
[[0 0 0]
 [0 1 0]]
>>> x = mindspore.tensor([1, 0, 2, 0, 3], mindspore.int32)
>>> output = mindspore.ops.nonzero(x, as_tuple=False)
>>> print(output)
[[0]
 [2]
 [4]]
>>> x = mindspore.tensor([[[1,  0], [-5, 0]]], mindspore.int32)
>>> output = mindspore.ops.nonzero(x, as_tuple=True)
>>> print(output)
(Tensor(shape=[2], dtype=Int64, value=[0, 0]),
 Tensor(shape=[2], dtype=Int64, value=[0, 1]),
 Tensor(shape=[2], dtype=Int64, value=[0, 0]))
>>> x = mindspore.tensor([1, 0, 2, 0, 3], mindspore.int32)
>>> output = mindspore.ops.nonzero(x, as_tuple=True)
>>> print(output)
(Tensor(shape=[3], dtype=Int64, value=[0, 2, 4]), )
```

mindspore.ops.numel

mindspore.ops.**numel**(*input*)

Return the total number of elements in the tensor.

Parameters

input (Tensor) –The input tensor.

Returns

The total number of elements in tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[2, 2], [2, 2]], mindspore.float32)
>>> print(mindspore.ops.numel(input_x))
4
```

mindspore.ops.permute

mindspore.ops.**permute**(*input*, *axis*)

Permute the input tensor along the specified axis.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*tuple(int)*) –The axis in a specified order.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]],␣
↪mindspore.float32)
>>> input_perm = (0, 2, 1)
>>> print(mindspore.ops.permute(input_x, input_perm))
[[[ 1.  4.]
  [ 2.  5.]
  [ 3.  6.]]
 [[ 7. 10.]
  [ 8. 11.]
  [ 9. 12.]]]
```

mindspore.ops.population_count

mindspore.ops.**population_count**(*input_x*)

Calculate the number of 1 bits in the binary representation of each element in the input tensor.

Parameters

input_x (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([0, 1, 3], mindspore.int16)
>>> output = mindspore.ops.population_count(input_x)
>>> print(output)
[0 1 2]
```

mindspore.ops.rank

mindspore.ops.**rank**(*input_x*)

Return the rank of a tensor.

Parameters

input_x (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_tensor = mindspore.tensor([[2, 2], [2, 2]], mindspore.float32)
>>> output = mindspore.ops.rank(input_tensor)
>>> print(output)
2
>>> print(type(output))
<class 'int'>
```

mindspore.ops.repeat_elements

mindspore.ops.**repeat_elements**(*x*, *rep*, *axis=0*)

Repeat elements of a tensor along an axis, like *mindspore.numpy.repeat()* .

Note: It is recommended to use :func:'mindspore.mint.repeat_interleave', the dimension of input 'x' can support a maximum of 8, and get better performance.

Parameters

- **x** (Tensor) −The input tensor.
- **rep** (*int*) −The number of times to repeat, must be positive.
- **axis** (*int*) −The axis along which to repeat. Default 0.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1 : repeat on axis 0
>>> x = mindspore.tensor([[0, 1, 2], [3, 4, 5]], mindspore.int32)
>>> output = mindspore.ops.repeat_elements(x, rep = 2, axis = 0)
>>> print(output)
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
>>> # case 2 : repeat on axis 1
>>> x = mindspore.tensor([[0, 1, 2], [3, 4, 5]], mindspore.int32)
>>> output = mindspore.ops.repeat_elements(x, rep = 2, axis = 1)
>>> print(output)
[[0 0 1 1 2 2]
 [3 3 4 4 5 5]]
```

mindspore.ops.repeat_interleave

mindspore.ops.**repeat_interleave**(*input*, *repeats*, *axis=None*)

Repeat elements of a tensor along an axis, like *mindspore.numpy.repeat()*.

Parameters

- **input** (Tensor) −The input tensor.
- **repeats** (*Union[int, tuple, list, Tensor]*) −The number of times to repeat, must be positive.
- **axis** (*int, optional*) −The axis along which to repeat, Default None. if dims is None, both the input and output tensors will be flattened into 1-D.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>>  # case 1 : repeat on axis 0
>>> input = mindspore.tensor([[0, 1, 2], [3, 4, 5]], mindspore.int32)
>>> output = mindspore.ops.repeat_interleave(input, repeats=2, axis=0)
>>> print(output)
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
>>>  # case 2 : repeat on axis 1
>>> input = mindspore.tensor([[0, 1, 2], [3, 4, 5]], mindspore.int32)
>>> output = mindspore.ops.repeat_interleave(input, repeats=2, axis=1)
>>> print(output)
[[0 0 1 1 2 2]
[3 3 4 4 5 5]]
```

mindspore.ops.reshape

`mindspore.ops.`**`reshape`**(*input*, *shape*)

Reshape the input tensor based on the given shape.

Note: The -1 in the parameter *shape* indicates that the size of that dimension is inferred from the other dimensions and the total number of elements in input tensor.

Parameters

- **`input`** (`Tensor`) –The input tensor.
- **`shape`** (*`Union[tuple[int], list[int], Tensor[int]]`*) –New shape.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> # case1: Parameter `shape` does not contain -1.
>>> output = mindspore.ops.reshape(input, (3, 2))
>>> print(output)
[[-0.1  0.3]
 [ 3.6  0.4]
 [ 0.5 -3.2]]
>>> # case2: Parameter `shape` contains -1.
>>> output = mindspore.ops.reshape(input, (-1, 6))
>>> print(output)
[[-0.1  0.3  3.6  0.4  0.5 -3.2]]
```

mindspore.ops.reverse_sequence

mindspore.ops.**reverse_sequence**(*x*, *seq_lengths*, *seq_dim*, *batch_dim=0*)

Partially reverse the input sequence.

Parameters

- **x** (Tensor) –The input tensor.
- **seq_lengths** (Tensor) –The specified reversing length.
- **seq_dim** (*int*) –The specified dimension for reversal.
- **batch_dim** (*int*) –The specified slice dimension. Default 0 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.float32)
>>> seq_lengths = mindspore.tensor([1, 2, 3])
>>> output = mindspore.ops.reverse_sequence(x, seq_lengths, seq_dim=1)
>>> print(output)
[[1. 2. 3.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.float32)
>>> seq_lengths = mindspore.tensor([1, 2, 3])
>>> output = mindspore.ops.reverse_sequence(x, seq_lengths, seq_dim=0, batch_dim=1)
>>> print(output)
[[1. 5. 9.]
 [4. 2. 6.]
 [7. 8. 3.]]
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.float32)
>>> seq_lengths = mindspore.tensor([2, 2, 3])
```

(continues on next page)

(continued from previous page)

```
>>> output = mindspore.ops.reverse_sequence(x, seq_lengths, seq_dim=1)
>>> print(output)
[[2. 1. 3.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.float32)
>>> seq_lengths = mindspore.tensor([3, 2, 3])
>>> output = mindspore.ops.reverse_sequence(x, seq_lengths, seq_dim=1)
>>> print(output)
[[3. 2. 1.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = mindspore.tensor([[1, 2, 3, 4], [5, 6, 7, 8]], mindspore.float32)
>>> seq_lengths = mindspore.tensor([4, 4])
>>> output = mindspore.ops.reverse_sequence(x, seq_lengths, seq_dim=1)
>>> print(output)
[[4. 3. 2. 1.]
 [8. 7. 6. 5.]]
```

mindspore.ops.roll

mindspore.ops.**roll**(*input*, *shifts*, *dims=None*)

Roll the elements of a tensor along a dimension.

Parameters

- **input** (Tensor) – The input tensor.
- **shifts** (*Union[list(int), tuple(int), int]*) – The amount of element shifting.
- **dims** (*Union[list(int), tuple(int), int], optional*) – Specify the dimension to move. Default None , which means the input tensor will be flattened before computation, and the result will be reshaped back to the original input shape.

Returns

Tensor

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0, 1, 2, 3, 4], mindspore.float32)
>>> # case1: Parameter `shifts` is positive
>>> output = mindspore.ops.roll(input, shifts=2, dims=0)
>>> print(output)
[3. 4. 0. 1. 2.]
>>> # case2: Parameter `shifts` is negative
>>> output = mindspore.ops.roll(input, shifts=-2, dims=0)
>>> print(output)
[2. 3. 4. 0. 1.]
```

mindspore.ops.row_stack

mindspore.ops.**row_stack**(*tensors*)

Alias for *mindspore.ops.vstack()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.scatter

mindspore.ops.**scatter**(*input*, *axis*, *index*, *src*)

Update the value in *src* to *input* according to the specified index. Refer to *mindspore.ops.tensor_scatter_elements()* for more details.

Note: If *src* is a tensor, the backward is supported only for the case *src.shape == index.shape*.

Parameters

- **input** (Tensor) −The input tensor.
- **axis** (*int*) −The axis to do update operation.
- **index** (Tensor) −The index to do update operation.
- **src** (Tensor, *float*) −The data to do the update operation with *input* .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3, 4, 5]], dtype=mindspore.float32)
>>> src = mindspore.tensor([[8, 8]], dtype=mindspore.float32)
>>> index = mindspore.tensor([[2, 4]], dtype=mindspore.int64)
>>> out = mindspore.ops.scatter(input=input, axis=1, index=index, src=src)
>>> print(out)
[[1. 2. 8. 4. 8.]]
>>> input = mindspore.tensor(mindspore.ops.zeros((5, 5)), dtype=mindspore.float32)
>>> src = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=mindspore.float32)
>>> index = mindspore.tensor([[0, 0, 0], [2, 2, 2], [4, 4, 4]], dtype=mindspore.int64)
>>> out = mindspore.ops.scatter(input=input, axis=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
[0. 0. 0. 0. 0.]
[4. 5. 6. 0. 0.]
[0. 0. 0. 0. 0.]
[7. 8. 9. 0. 0.]]
>>> input = mindspore.tensor(mindspore.ops.zeros((5, 5)), dtype=mindspore.float32)
>>> src = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> index = mindspore.tensor([[0, 2, 4], [0, 2, 4], [0, 2, 4]], dtype=mindspore.int64)
>>> out = mindspore.ops.scatter(input=input, axis=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
[4. 0. 5. 0. 6.]
[7. 0. 8. 0. 9.]
[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]]
```

mindspore.ops.scatter_nd

mindspore.ops.**scatter_nd**(*indices*, *updates*, *shape*)

Scatters a tensor into a new tensor depending on the specified indices.

Creates an empty tensor with the given *shape*, and set values by scattering the update tensor depending on indices. The empty tensor has rank P and *indices* has rank Q.

The *shape* is $(s_0, s_1, ..., s_{P-1})$, where $P \geq 1$.

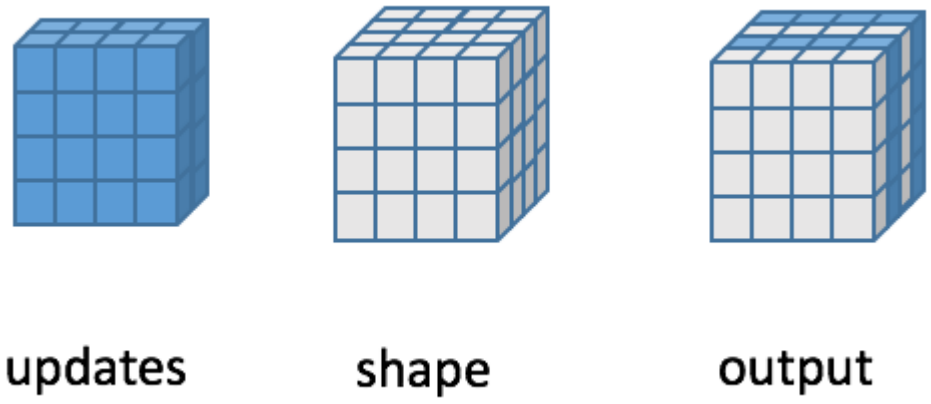
indices has shape $(i_0, i_1, ..., i_{Q-2}, N)$, where $Q \geq 2$ and $N \leq P$.

The last dimension of *indices* (with length N) indicates slices along the N th dimension of the empty tensor.

updates is a tensor of rank $Q - 1 + P - N$, and its shape is $(i_0, i_1, ..., i_{Q-2}, s_N, s_{N+1}, ..., s_{P-1})$.

If *indices* contains duplicates, the duplicate *updates* are summed.

The following figure shows the calculation process of inserting two new value matrices into the first dimension with rank-3:



Parameters

- **indices** (Tensor) −The index of scattering in the new tensor. The rank of *indices* must be at least 2 and *indices.shape[-1]* <= *len(shape)*.
- **updates** (Tensor) −The source tensor to be updated. It has shape *indices.shape[:-1] + shape[indices.shape[-1]:]*.
- **shape** (*tuple[int]*) −The shape of the output tensor. *shape* can not be empty, and the elements in *shape* must be greater than or equal to 1.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2],
...                              [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[1, 1, 1, 1], [2, 2, 2, 2],
...                              [3, 3, 3, 3], [4, 4, 4, 4]]], mindspore.float32)
>>> shape = (4, 4, 4)
>>> output = mindspore.ops.scatter_nd(indices, updates, shape)
>>> print(output)
[[[1. 1. 1. 1.]
  [2. 2. 2. 2.]
  [3. 3. 3. 3.]
  [4. 4. 4. 4.]]
 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]
 [[1. 1. 1. 1.]
  [2. 2. 2. 2.]
  [3. 3. 3. 3.]
  [4. 4. 4. 4.]]
 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
>>> indices = mindspore.tensor([[0, 1], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([3.2, 1.1], mindspore.float32)
>>> shape = (3, 3)
>>> output = mindspore.ops.scatter_nd(indices, updates, shape)
>>> # In order to facilitate understanding, explain the operator pseudo-operation process␣
↪step by step:
>>> # Step 1: Generate an empty Tensor of the specified shape according to the shape
>>> # [
>>> #     [0. 0. 0.]
>>> #     [0. 0. 0.]
>>> #     [0. 0. 0.]
>>> # ]
>>> # Step 2: Modify the data at the specified location according to the indicators
>>> # 0th row of indices is [0, 1], 0th row of updates is 3.2.
>>> # means that the empty tensor in the 0th row and 1st col set to 3.2
>>> # [
>>> #     [0. 3.2. 0.]
>>> #     [0. 0.   0.]
>>> #     [0. 0.   0.]
>>> # ]
>>> # 1th row of indices is [1, 1], 1th row of updates is 1.1.
>>> # means that the empty tensor in the 1th row and 1st col set to 1.1
>>> # [
>>> #     [0. 3.2. 0.]
>>> #     [0. 1.1  0.]
>>> #     [0. 0.   0.]
```

(continues on next page)

(continued from previous page)

```
>>> # ]
>>> # The final result is as follows:
>>> print(output)
[[0. 3.2 0.]
 [0. 1.1 0.]
 [0. 0.  0.]]
```

mindspore.ops.select

mindspore.ops.**select**(*condition*, *input*, *other*)

The conditional tensor determines whether the corresponding element in the output must be selected from *input* (if True) or *other* (if False) based on the value of each element.

It can be defined as:

$$out_i = \begin{cases} input_i, & \text{if } condition_i \\ other_i, & \text{otherwise} \end{cases}$$

Parameters

- **condition** (Tensor[bool]) – The condition tensor.
- **input** (Union[Tensor, int, float]) – The first tensor or number to be selected.
- **other** (Union[Tensor, int, float]) – The second tensor or number to be selected.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case1: Both `input` and `other` are tensor
>>> cond = mindspore.tensor([True, False])
>>> x = mindspore.tensor([2,3], mindspore.float32)
>>> y = mindspore.tensor([1,2], mindspore.float32)
>>> output1 = mindspore.ops.select(cond, x, y)
>>> print(output1)
[2. 2.]
>>> # case2: Both `input` and `other` are number
>>> output2 = mindspore.ops.select(cond, input=1, other=2)
>>> print(output2)
[1 2]
>>> # case3: `input` is tensor and `other` is number
>>> output3 = mindspore.ops.select(cond, x, other=3)
>>> print(output3)
[2. 3.]
```

mindspore.ops.select_scatter

mindspore.ops.**select_scatter**(*input*, *src*, *axis*, *index*)

On the specified dimension *axis* of *input* , *src* is scattered into *input* on the specified *index* of *input* .

Parameters

- **input** (Tensor) −The input tensor.
- **src** (Tensor) −The source tensor.
- **axis** (*int*) −The dimension of *input* to be embedded.
- **index** (*int*) −The location of scattering on the specified dimension.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.zeros((2, 3, 3))
>>> src = mindspore.ops.ones((2, 3))
>>> output = mindspore.ops.select_scatter(input, src, axis=1, index=1)
>>> print(output)
[[[0. 0. 0.]
  [1. 1. 1.]
  [0. 0. 0.]]
 [[0. 0. 0.]
  [1. 1. 1.]
  [0. 0. 0.]]]
```

mindspore.ops.sequence_mask

mindspore.ops.**sequence_mask**(*lengths*, *maxlen=None*)

Returns a mask tensor representing the first N positions of each cell.

If *lengths* has shape $(d_1, d_2, ..., d_n)$, then the resulting tensor mask has type and shape $(d_1, d_2, ..., d_n, maxlen)$, with mask $[i_1, i_2, ..., i_n, j] = (j < lengths[i_1, i_2, ..., i_n])$.

Parameters

- **lengths** (Tensor) −Tensor to calculate the mask for. All values in this tensor should be less than or equal to *maxlen*. Values greater than *maxlen* will be treated as *maxlen*.
- **maxlen** (*int*) −size of the last dimension of returned tensor. Must be positive and same type as elements in *lengths*. Default None .

Returns

Tensor, shape is *lengths.shape + (maxlen,)* .

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: When maxlen is assigned
>>> x = mindspore.tensor([1, 2, 3, 4])
>>> output = mindspore.ops.sequence_mask(x, 5)
>>> print(output)
[[ True False False False False]
 [ True  True False False False]
 [ True  True  True False False]
 [ True  True  True  True False]]
>>> # case 2: When there is 0 in x
>>> x = mindspore.tensor([[1, 3], [2, 0]])
>>> output = mindspore.ops.sequence_mask(x, 5)
>>> print(output)
[[[ True False False False False]
  [ True  True  True False False]]
 [[ True  True False False False]
  [False False False False False]]]
>>> # case 3: when the maxlen is not assigned
>>> x = mindspore.tensor([[1, 3], [2, 4]])
>>> output = mindspore.ops.sequence_mask(x)
>>> print(output)
[[[ True False False False ]
  [ True  True  True False ]]
 [[ True  True False False ]
  [ True  True  True  True ]]]
```

mindspore.ops.shape

mindspore.ops.**shape**(*input_x*)

Return the shape of the input tensor.

Parameters

input_x (Tensor) – The input tensor.

Returns

tuple[int]

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.ops.ones(shape=[3, 2, 1])
>>> output = mindspore.ops.shape(input_x)
>>> print(output)
(3, 2, 1)
```

mindspore.ops.shuffle

mindspore.ops.**shuffle**(*x*, *seed=None*)

Randomly shuffle a tensor along its first dimension.

Parameters

- **x** (Tensor) −The input tensor.
- **seed** (*int, optional*) −Random seed. Default None , which is equivalent to 0.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3, 4], mindspore.float32)
>>> output = mindspore.ops.shuffle(x, seed=1)
>>> print(output)
[3. 4. 2. 1.]
```

mindspore.ops.size

mindspore.ops.**size**(*input_x*)

Count the total number of elements in *input_x* .

Parameters

input_x (Tensor) −The input tensor.

Returns

int

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[2, 2], [2, 2]], mindspore.float32)
>>> output = mindspore.ops.size(input_x)
>>> print(output)
4
```

mindspore.ops.slice

mindspore.ops.**slice**(*input_x*, *begin*, *size*)

Slice a tensor in the specified shape.

Note: *begin* is zero-based and *size* is one-based.

If *size[i]* is -1, all remaining elements in dimension i are included in the slice. This is equivalent to setting $size[i] = input\_x.shape(i) - begin[i]$.

Parameters

- **input_x** (Tensor) –The input tensor.
- **begin** (*Union[tuple, list]*) –The beginning of the slice which represents the offset in each dimension.
- **size** (*Union[tuple, list]*) –The size of the slice.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> data = mindspore.tensor([[[1, 1, 1], [2, 2, 2]],
...                          [[3, 3, 3], [4, 4, 4]],
...                          [[5, 5, 5], [6, 6, 6]]], mindspore.int32)
>>> output = mindspore.ops.slice(data, (1, 0, 0), (1, 1, 3))
>>> print(output)
[[[3 3 3]]]
>>> output = mindspore.ops.slice(data, (1, 0, 0), (1, 1, 2))
>>> print(output)
[[[3 3]]]
>>> output = mindspore.ops.slice(data, (1, 0, 0), (1, 1, 1))
>>> print(output)
[[[3]]]
>>> output = mindspore.ops.slice(data, (1, 1, 0), (1, 1, 3))
>>> print(output)
[[[4 4 4]]]
>>> output = mindspore.ops.slice(data, (1, 0, 1), (1, 1, 2))
>>> print(output)
[[[3 3]]]
```

mindspore.ops.slice_scatter

mindspore.ops.**slice_scatter**(*input*, *src*, *axis=0*, *start=None*, *end=None*, *step=1*)

Embed *src* into the sliced *input* along the specified *axis* .

Parameters

- **input** (Tensor) –The input tensor.
- **src** (Tensor) –The source tensor to be embedded into *input* .
- **axis** (*int, optional*) –The axis of *input* to be sliced. Default 0 .
- **start** (*int, optional*) –The start index for embedding in the specified axis. Default None , which means *start* is 0 .
- **end** (*int, optional*) –The end index for embedding in the specified axis. Default None , which means *end* is the length of *input* in the specified axis.
- **step** (*int, optional*) –The step size to skip during embedding. Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> a = mindspore.ops.zeros((4, 6))
>>> b = mindspore.ops.ones((4, 3))
>>> output = mindspore.ops.slice_scatter(input=a, src=b, axis=1, start=0, end=5, step=2)
>>> print(output)
[[1. 0. 1. 0. 1. 0.]
 [1. 0. 1. 0. 1. 0.]
 [1. 0. 1. 0. 1. 0.]
 [1. 0. 1. 0. 1. 0.]]
```

mindspore.ops.sort

mindspore.ops.**sort**(*input_x*, *axis=- 1*, *descending=False*)

Sort the elements of the input tensor along the given axis.

Note: The Ascend backend only supports sorting the last dimension.

Parameters

- **input_x** (Tensor) –The input tensor.
- **axis** (*int, optional*) –The axis to sort along. Default -1 , means the last dimension.
- **descending** (*bool, optional*) –Sorting method. True means the elements are sorted in descending order, or else sorted in ascending order. Default False .

Warning: Currently, the data types of float16, uint8, int8, int16, int32, int64 are well supported. If use float32, it may cause loss of accuracy.

Returns

Tuple(sorted_tensor, indices) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[8, 2, 1], [5, 9, 3], [4, 6, 7]], mindspore.float16)
>>> output = mindspore.ops.sort(x)
>>> # The output below is based on the Ascend platform.
>>> print(output)
(Tensor(shape=[3, 3], dtype=Float16, value=
[[ 1.0000e+00,  2.0000e+00,  8.0000e+00],
[ 3.0000e+00,  5.0000e+00,  9.0000e+00],
[ 4.0000e+00,  6.0000e+00,  7.0000e+00]]), Tensor(shape=[3, 3], dtype=Int32, value=
[[2, 1, 0],
[2, 0, 1],
[0, 1, 2]]))
```

mindspore.ops.space_to_batch_nd

`mindspore.ops.`**`space_to_batch_nd`**(*input_x*, *block_size*, *paddings*)

Divides a tensor's spatial dimensions into blocks and combines the block sizes with the original batch.

$$n' = n * (block\_size[0] * ... * block\_size[M])$$
$$w'_i = (w_i + paddings[i][0] + paddings[i][1])//block\_size[i]$$

Note:

- This operation divides the spatial dimensions [1, ⋯, M] of the input into blocks of size *block_size* and interleaves them into the batch dimension (default: dimension 0). Before splitting, the spatial dimensions are padded with zeros according to *paddings*.
- If the input shape is $(n, c_1, ...c_k, w_1, ..., w_M)$, then the output shape will be $(n', c_1, ...c_k, w'_1, ..., w'_M)$.
- If *block_size* is a tuple or list, the length of *block_size* is M corresponding to the number of spatial dimensions. If *block_size* is an int, the block size of M dimensions are the same, equal to *block_size*. M must be 2 on Ascend.

Parameters

- **`input_x`** (`Tensor`) −The input tensor, must be a 4-D tensor on Ascend.
- **`block_size`** (*`Union[list(int), tuple(int), int]`*) −Specifies the block size for spatial dimension division.
- **`paddings`** (*`Union[tuple, list]`*) −The padding size for each spatial dimension.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> block_size = [2, 2]
>>> paddings = [[0, 0], [0, 0]]
>>> input_x = mindspore.tensor([[[[1, 2], [3, 4]]]], mindspore.float32)
>>> output = mindspore.ops.space_to_batch_nd(input_x, block_size, paddings)
>>> print(output)
[[[[1.]]]
 [[[2.]]]
 [[[3.]]]
 [[[4.]]]]
```

mindspore.ops.sparse_segment_mean

mindspore.ops.**sparse_segment_mean**(*x*, *indices*, *segment_ids*)

Computes the mean of sparse segments in the input tensor.

$$output_i = \frac{\sum_j x_{indices[j]}}{N}$$

where N is the number of elements where $segment\_ids[j] == i$. If *segment_ids* doesn't contain *i*, then $output[i] = 0$.

Note:

- On CPU, values in *segment_ids* must be sorted and *indices* must be within range[0, x.shape[0]).
- On GPU, unsorted *segment_ids* may result in undefined but safe behavior.Out-of-range *indices* will be ignored.

Parameters

- **x** (Tensor) −The input tensor with at least one dimension.
- **indices** (Tensor) −The specified indices, a 1-D tensor.
- **segment_ids** (Tensor) −A 1-D tensor, must be sorted and can contain duplicates.

Returns

Tensor

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[0, 1, 2], [1, 2, 3], [3, 6, 7]], dtype=mindspore.float32)
>>> indices = mindspore.tensor([0, 1, 2], dtype=mindspore.int32)
>>> segment_ids = mindspore.tensor([1,2,2], dtype=mindspore.int32)
>>> out = mindspore.ops.sparse_segment_mean(x, indices, segment_ids)
>>> print(out)
[[0. 0. 0.]
 [0. 1. 2.]
 [2. 4. 5.]]
```

mindspore.ops.split

mindspore.ops.**split**(*tensor*, *split_size_or_sections*, *axis=0*)

Split the tensor into chunks along the given axis.

Parameters

- **tensor** (Tensor) –The input tensor.
- **split_size_or_sections** (*Union[int, tuple(int), list(int)]*) –The size of chunks after splited.
- **axis** (*int, optional*) –The axis along which to split. Default 0 .

Note:

- If *split_size_or_sections* is an int type, the input tensor will be evenly divided into chunks of size *split_size_or_sections* . The last chunk will have a size equal to the remainder if tensor.shape[axis] is not divisible by *split_size_or_sections* .
- If *split_size_or_sections* is a tuple or list, *tensor* will be split along *axis* into len(*split_size_or_sections*) chunks with sizes specified by *split_size_or_sections* .

Returns

Tuple of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case1: `split_size_or_sections` is an int type
>>> input_x = mindspore.ops.arange(10).astype("float32")
>>> output = mindspore.ops.split(tensor=input_x, split_size_or_sections=3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value=[0.00000000e+00, 1.00000000e+00, 2.00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value=[3.00000000e+00, 4.00000000e+00, 5.00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value=[6.00000000e+00, 7.00000000e+00, 8.00000000e+00]),
 Tensor(shape=[1], dtype=Float32, value=[9.00000000e+00]))
>>> # case2: `split_size_or_sections` is a list type
>>> output = mindspore.ops.split(tensor=input_x, split_size_or_sections=[3, 3, 4])
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
(Tensor(shape=[3], dtype=Float32, value=[0.00000000e+00, 1.00000000e+00, 2.00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value=[3.00000000e+00, 4.00000000e+00, 5.00000000e+00]),
 Tensor(shape=[4], dtype=Float32, value=[6.00000000e+00, 7.00000000e+00, 8.00000000e+00, 9.
↪00000000e+00]))
```

mindspore.ops.squeeze

mindspore.ops.**squeeze**(*input*, *axis=None*)

Remove length one axes from input tensor.

Note:

- Please note that in dynamic graph mode, the output tensor will share data with the input tensor, and there is no Tensor data copy process.
- The dimension index starts at 0 and must be in the range *[-input.ndim, input.ndim]*.
- In GE mode, only support remove dimensions of size 1 from the shape of input tensor.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*Union[int, tuple(int), list(int)]*) –The axis to be removed. Default: None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.ones(shape=[3, 2, 1])
>>> output = mindspore.ops.squeeze(input)
>>> print(output)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

mindspore.ops.stack

mindspore.ops.**stack**(*tensors*, *axis=0*)

Stack input tensors in specified axis.

Parameters

- **tensors** (*Union[tuple, list]*) – The input tensors.
- **axis** (*int*) – Axis to stack. Default `0` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x1 = mindspore.tensor([0, 1], mindspore.float32)
>>> input_x2 = mindspore.tensor([2, 3], mindspore.float32)
>>> output = mindspore.ops.stack((input_x1, input_x2), 0)
>>> print(output)
[[0. 1.]
 [2. 3.]]
```

mindspore.ops.strided_slice

mindspore.ops.**strided_slice**(*input_x*, *begin*, *end*, *strides*, *begin_mask=0*, *end_mask=0*, *ellipsis_mask=0*, *new_axis_mask=0*, *shrink_axis_mask=0*)

Extracts a strided slice of a Tensor based on *begin/end* index and *strides*.

Note:

- *begin* , *end* and *strides* must have the same shape.
- *begin* , *end* and *strides* are all 1-D Tensor, and their shape size must not greater than the dim of *input_x*.

Example: For *input_x* with shape $(5, 6, 7)$, set *begin*, *end* and *strides* to (1, 3, 2), (3, 5, 6), (1, 1, 2) respectively, then elements from index 1 to 3 are extracted for dim 0, index 3 to 5 are extracted for dim 1 and index 2 to 6 with a *stirded* of 2 are extracted for dim 2, this process is equivalent to a pythonic slice *input_x[1:3, 3:5, 2:6:2]*.

If the length of *begin*, *end* and *strides* is smaller than the dim of *input_x*, then all elements are extracted from the missing dims, it behaves like all the missing dims are filled with zeros, size of that missing dim and ones.

Example: For Tensor *input_x* with shape $(5, 6, 7)$, set *begin*, *end* and *strides* to (1, 3), (3, 5), (1, 1) respectively, then elements from index 1 to 3 are extracted for dim 0, index 3 to 5 are extracted for dim 1 and index 3 to 5 are extracted for dim 2, this process is equivalent to a pythonic slice *input_x[1:3, 3:5, 0:7]*.

Here's how a mask works: For each specific mask, it will be converted to a binary representation internally, and then reverse the result to start the calculation. For Tensor *input_x* with shape $(5, 6, 7)$. Given mask value of 3 which can be represented as 0b011. Reverse that we get 0b110, which implies the first and second dim of the original Tensor will be effected by this mask. See examples below, for simplicity all mask mentioned below are all in their reverted binary form:

- *begin_mask* and *end_mask*

  If the ith bit of *begin_mask* is 1, *begin[i]* is ignored and the fullest possible range in that dimension is used instead. *end_mask* is analogous, except with the end range. For Tensor *input_x* with shape $(5, 6, 7, 8)$, if *begin_mask* is 0b110, *end_mask* is 0b011, the slice *input_x[0:3, 0:6, 2:7:2]* is produced.

- *ellipsis_mask*

  If the ith bit of *ellipsis_mask* is 1, as many unspecified dimensions as needed will be inserted between other dimensions. Only one non-zero bit is allowed in *ellipsis_mask*. For Tensor *input_x* with shape $(5, 6, 7, 8)$, *input_x[2:,⋯,:6]* is equivalent to *input_x[2:5,:,:,0:6]* , *input_x[2:,⋯]* is equivalent to *input_x[2:5,:,:,:]*.

- *new_axis_mask*

  If the ith bit of *new_axis_mask* is 1, *begin*, *end* and *strides* are ignored and a new length 1 dimension is added at the specified position in the output Tensor. For Tensor *input_x* with shape $(5, 6, 7)$, if *new_axis_mask* is 0b110, a new dim is added to the second dim, which will produce a Tensor with shape $(5, 1, 6, 7)$.

- *shrink_axis_mask*

  If the ith bit of *shrink_axis_mask* is 1, *begin*, *end* and *strides* are ignored and dimension i will be shrunk to 0. For Tensor *input_x* with shape $(5, 6, 7)$, if *shrink_axis_mask* is 0b010, it is equivalent to slice *x[:, 5, :]* and results in an output shape of $(5, 7)$.

Note: *new_axis_mask* and *shrink_axis_mask* are not recommended to use at the same time, it might incur unexpected result.

Parameters

- **input_x** (Tensor) −The input tensor.
- **begin** (*tuple[int]*) −Specify the index to start slicing.
- **end** (*tuple[int]*) −Specify the index to end slicing.
- **strides** (*tuple[int]*) −Specify the step size for slicing in each dimension.
- **begin_mask** (*int, optional*) −Starting index of the slice. Default `0` .
- **end_mask** (*int, optional*) −Ending index of the slice. Default `0` .
- **ellipsis_mask** (*int, optional*) −An int mask, ignore slicing operation when set to 1. Default `0` .
- **new_axis_mask** (*int, optional*) −An int mask for adding new dims. Default `0` .
- **shrink_axis_mask** (*int, optional*) −An int mask for shrinking dims. Default `0` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[[1, 1, 1],
...                            [2, 2, 2]],
...                           [[3, 3, 3],
...                            [4, 4, 4]],
...                           [[5, 5, 5],
...                            [6, 6, 6]]])
>>> mindspore.ops.strided_slice(input, [1, 0, 0], [2, 1, 3], [1, 1, 1])
Tensor(shape=[1, 1, 3], dtype=Int64, value=
[[[3, 3, 3]]])
>>> mindspore.ops.strided_slice(input, [1, 0, 0], [2, 2, 3], [1, 1, 1])
Tensor(shape=[1, 2, 3], dtype=Int64, value=
[[[3, 3, 3],
  [4, 4, 4]]])
>>> mindspore.ops.strided_slice(input, [1, -1, 0], [2, -3, 3], [1, -1, 1])
Tensor(shape=[1, 2, 3], dtype=Int64, value=
[[[4, 4, 4],
  [3, 3, 3]]])
```

mindspore.ops.sum

mindspore.ops.**sum**(*input*, *dim=None*, *keepdim=False*, *, *dtype=None*)

Calculate sum of tensor elements over a given dim.

Note: The *dim* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*Union[None, int, tuple(int), list(int), Tensor]*) –Dimensions along which the sum is calculated. Default None .
- **keepdim** (*bool*) –Whether the output tensor has dim retained or not. If True , keep these reduced dimensions and the length is 1. If False , will not keep these dimensions. Default False .

Note:

- If *dim* is None , sum is calculated on all the elements of the input tensor.
- If *dim* is a tuple or list of ints or tensor, sum is calculated on all dimensions specified in *dim* .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                       [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                       [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]],␣
↪mindspore.float32)
>>> out = mindspore.ops.sum(input=x)
>>> print(out)
270.0
>>> out = mindspore.ops.sum(input=x, dim=1)
>>> print(out)
[[ 6.  6.  6.  6.  6.  6.]
 [15. 15. 15. 15. 15. 15.]
 [24. 24. 24. 24. 24. 24.]]
>>> out = mindspore.ops.sum(input=x, dim=2)
>>> print(out)
[[ 6. 12. 18.]
 [24. 30. 36.]
 [42. 48. 54.]]
>>> out = mindspore.ops.sum(input=x, dim=[1, 2])
>>> print(out)
[ 36.  90. 144.]
>>> out = mindspore.ops.sum(input=x, dim=2, keepdim=True)
>>> print(out)
[[[ 6.]
  [12.]
  [18.]]
 [[24.]
  [30.]
  [36.]]
 [[42.]
  [48.]
  [54.]]]
>>> print(out.ndim)
3
```

mindspore.ops.swapaxes

mindspore.ops.**swapaxes**(*input*, *axis0*, *axis1*)

Interchange two axes of a tensor.

Parameters

- **input** (Tensor) – The input tensor.
- **axis0** (*int*) – First axis.
- **axis1** (*int*) – Second axis.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.ones(([2, 3, 4])))
>>> output = mindspore.ops.swapaxes(input, 0, 2)
>>> print(output.shape)
(4, 3, 2)
```

mindspore.ops.swapdims

mindspore.ops.**swapdims**(*input*, *dim0*, *dim1*)

Interchange two dims of a tensor. This function is equivalent to *mindspore.ops.swapaxes()* function.

Parameters

- **input** (Tensor) –The input tensor.
- **dim0** (*int*) –First dim.
- **dim1** (*int*) –Second dim.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.ones([2, 3, 4]))
>>> output = mindspore.ops.swapdims(input, 0, 2)
>>> print(output.shape)
(4, 3, 2)
```

mindspore.ops.tensor_scatter_add

mindspore.ops.**tensor_scatter_add**(*input_x*, *indices*, *updates*)

Return a new tensor by adding the values from *updates* in *input_x* indicated by *indices* .

$$output\left[indices\right] = input\_x + update$$

Note:

- On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor.
- On CPU, if some values of the *indices* are out of bound, raising an index error.
- On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.

Parameters

- **input_x** (Tensor) −The input tensor.
- **indices** (Tensor) −The specified indices.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> indices = mindspore.tensor([[0, 0], [0, 0]], mindspore.int32)
>>> updates = mindspore.tensor([1.0, 2.2], mindspore.float32)
>>> output = mindspore.ops.tensor_scatter_add(input_x, indices, updates)
>>> print(output)
[[ 3.1  0.3  3.6]
 [ 0.4  0.5 -3.2]]
```

mindspore.ops.tensor_scatter_div

mindspore.ops.**tensor_scatter_div**(*input_x*, *indices*, *updates*)

Return a new tensor which *input_x* is divided by the values from *updates* indicated by *indices* .

$$output\,[indices] = input\_x \div update$$

Note:

- On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor.
- On CPU, if some values of the *indices* are out of bound, raising an index error.
- On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.
- The operator can't handle division by 0 exceptions, so the user needs to make sure there is no 0 value in *updates*.

Parameters

- **input_x** (Tensor) −The input tensor.
- **indices** (Tensor) −The specified indices.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> indices = mindspore.tensor([[0, 0], [0, 0]], mindspore.int32)
>>> updates = mindspore.tensor([1.0, 2.0], mindspore.float32)
>>> output = mindspore.ops.tensor_scatter_div(input_x, indices, updates)
>>> print(output)
[[-0.05  0.3  3.6  ]
 [ 0.4   0.5  -3.2 ]]
```

mindspore.ops.tensor_scatter_max

mindspore.ops.**tensor_scatter_max**(*input_x*, *indices*, *updates*)

Return a new tensor by performing a maximum update on *input_x* at the specified indices with the given update values.

$$output\,[indices] = \max(input\_x, update)$$

Note:

- On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor.
- On CPU, if some values of the *indices* are out of bound, raising an index error.
- On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.

Parameters

- **input_x** (Tensor) –The input tensor.
- **indices** (Tensor) –The specified indices. The rank must be at least 2.
- **updates** (Tensor) –The update values.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[1, 2, 3], [4, 5, 6]])
>>> indices = mindspore.tensor([[0, 0], [1, 1]])
>>> updates = mindspore.tensor([5, 5])
>>> mindspore.ops.tensor_scatter_max(input_x, indices, updates)
Tensor(shape=[2, 3], dtype=Int64, value=
[[5, 2, 3],
 [4, 5, 6]])
```

mindspore.ops.tensor_scatter_min

mindspore.ops.**tensor_scatter_min**(*input_x*, *indices*, *updates*)

Return a new tensor by performing a minimum update on *input_x* at the specified indices with the given update values.

$$output\left[indices\right] = \min(input\_x, update)$$

Note:

- On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor.
- On CPU, if some values of the *indices* are out of bound, raising an index error.
- On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.

Parameters

- **input_x** (Tensor) −The input tensor.
- **indices** (Tensor) −The specified indices. The rank must be at least 2.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[1, 2, 3], [4, 5, 6]], mindspore.float32)
>>> indices = mindspore.tensor([[0, 0], [1, 1]])
>>> updates = mindspore.tensor([5, 1], mindspore.float32)
>>> mindspore.ops.tensor_scatter_min(input_x, indices, updates)
Tensor(shape=[2, 3], dtype=Float32, value=
[[ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00],
 [ 4.00000000e+00,  1.00000000e+00,  6.00000000e+00]])
```

mindspore.ops.tensor_scatter_mul

mindspore.ops.**tensor_scatter_mul**(*input_x*, *indices*, *updates*)

Return a new tensor by performing a multiplication update on *input_x* at the specified indices with the given update values.

$$output\left[indices\right] = input\_x \times update$$

Note:

- If some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to *input_x*.

Parameters

- **input_x** (Tensor) −The input tensor.
- **indices** (Tensor) −The specified indices. The rank must be at least 2.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[1, 2, 3], [4, 5, 6]])
>>> indices = mindspore.tensor([[0, 0], [1, 1]])
>>> updates = mindspore.tensor([5, 5])
>>> mindspore.ops.tensor_scatter_mul(input_x, indices, updates)
Tensor(shape=[2, 3], dtype=Int64, value=
[[ 5,  2,  3],
 [ 4, 25,  6]])
```

mindspore.ops.tensor_scatter_sub

mindspore.ops.**tensor_scatter_sub**(*input_x*, *indices*, *updates*)

Return a new tensor by performing a subtraction update on *input_x* at the specified indices with the given update values.

$$output\left[indices\right] = input\_x - update$$

Note: On GPU, if some values of the *indices* are out of bound, instead of raising an index error, the corresponding *updates* will not be updated to self tensor. On CPU, if some values of the *indices* are out of bound, raising an index error. On Ascend, out of bound checking is not supported, if some values of the *indices* are out of bound, unknown errors may be caused.

Parameters

- **input_x** (Tensor) −The input tensor.
- **indices** (Tensor) −The specified indices. The rank must be at least 2.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3], [4, 5, 6]], mindspore.float32)
>>> indices = mindspore.tensor([[0, 0], [1, 1]])
>>> updates = mindspore.tensor([5, 5], mindspore.float32)
>>> mindspore.ops.tensor_scatter_sub(input, indices, updates)
Tensor(shape=[2, 3], dtype=Float32, value=
[[-4.00000000e+00,  2.00000000e+00,  3.00000000e+00],
 [ 4.00000000e+00,  0.00000000e+00,  6.00000000e+00]])
```

mindspore.ops.tensor_scatter_elements

mindspore.ops.**tensor_scatter_elements**(*input_x*, *indices*, *updates*, *axis=0*, *reduction='none'*)

Return a new tensor by performing a specified operation update on *input_x* at the specified indices with the given update values.

Not support implicit type conversion.

For example: the output of a 3-D tensor is

```
output[indices[i][j][k]][j][k] = updates[i][j][k]  # if axis == 0, reduction == "none"

output[i][indices[i][j][k]][k] += updates[i][j][k]  # if axis == 1, reduction == "add"

output[i][j][indices[i][j][k]] = updates[i][j][k]  # if axis == 2, reduction == "none"
```

Warning:

- The order in which updates are applied is nondeterministic, meaning that if there are multiple index vectors in *indices* that correspond to the same position, the value of that position in the output will be nondeterministic.
- On Ascend, the reduction only support set to "none" for now.
- On Ascend, the data type of *input_x* must be float16 or float32.
- This is an experimental API that is subject to change or deletion.

Note: If some values of the *indices* exceed the upper or lower bounds of the index of *input_x*, instead of raising an index error, the corresponding *updates* will not be updated to *input_x*. The backward is supported only for the case *updates.shape* == *indices.shape*.

Parameters

- **input_x** (Tensor) –The input tensor. The rank must be at least 1.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.
- **axis** (*int*) –The axis along which to index. Default 0.
- **reduction** (*str*) –The specified operation, supports none , add .
  - If none, *updates* will be assigned to *input_x* according to *indices*.
  - If add, *updates* will be added to *input_x* according to *indices*. Default none.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[1, 2, 3, 4, 5]])
>>> indices = mindspore.tensor([[2, 4]])
>>> updates = mindspore.tensor([[8, 8]])
>>> output = mindspore.ops.tensor_scatter_elements(input_x, indices, updates, axis=1,␣
↪reduction="none")
>>> print(output)
[[1 2 8 4 8]]
>>> output = mindspore.ops.tensor_scatter_elements(input_x, indices, updates, axis=1,␣
↪reduction="add")
>>> print(output)
[[ 1  2 11  4 13]]
```

mindspore.ops.tensor_split

mindspore.ops.**tensor_split**(*input*, *indices_or_sections*, *axis=0*)

Split the input tensor into multiple subtensors according to the specified indices or chunks.

Parameters

- **input** (Tensor) –The input tensor.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –The specified indices or chunks.
  - If it is an integer, input tensor will be split into *indices_or_sections* sections.
    - If $input.shape[axis]$ can be divisible by *indices_or_sections*, sub-sections will have equal size $input.shape[axis]/n$.
    - If $input.shape[axis]$ can not be divisible by *indices_or_sections*, the first $input.shape[axis] \bmod n$ sections will have size $input.shape[axis]//n + 1$, and the rest will have size $input.shape[axis]//n$.
  - If it is a tuple(int) or list(int) type, it represts indices and the input tensor will be split at the indices.
- **axis** (*int, optional*) –The axis along which to split. Default 0 .

Returns

Tuple of tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0, 1, 2, 3, 4, 5, 6, 7])
>>> mindspore.ops.tensor_split(input, 3)
(Tensor(shape=[3], dtype=Int64, value= [0, 1, 2]),
 Tensor(shape=[3], dtype=Int64, value= [3, 4, 5]),
 Tensor(shape=[2], dtype=Int64, value= [6, 7]))
>>> input = mindspore.tensor([0, 1, 2, 3, 4, 5, 6])
>>> mindspore.ops.tensor_split(input, 3)
(Tensor(shape=[3], dtype=Int64, value= [0, 1, 2]),
 Tensor(shape=[2], dtype=Int64, value= [3, 4]),
 Tensor(shape=[2], dtype=Int64, value= [5, 6]))
>>> mindspore.ops.tensor_split(input, (1, 6))
(Tensor(shape=[1], dtype=Int64, value= [0]),
 Tensor(shape=[5], dtype=Int64, value= [1, 2, 3, 4, 5]),
 Tensor(shape=[1], dtype=Int64, value= [6]))
>>> input = mindspore.tensor([[ 0,  1,  2,  3,  4,  5,  6],
...                           [ 7,  8,  9, 10, 11, 12, 13]])
>>> mindspore.ops.tensor_split(input, 3, axis=1)
(Tensor(shape=[2, 3], dtype=Int64, value=
 [[0, 1, 2],
  [7, 8, 9]]),
 Tensor(shape=[2, 2], dtype=Int64, value=
 [[ 3,  4],
  [10, 11]]),
 Tensor(shape=[2, 2], dtype=Int64, value=
 [[ 5,  6],
  [12, 13]]))
>>> mindspore.ops.tensor_split(input, (1, 6), axis=1)
(Tensor(shape=[2, 1], dtype=Int64, value=
 [[0],
  [7]]),
 Tensor(shape=[2, 5], dtype=Int64, value=
 [[ 1,  2,  3,  4,  5],
  [ 8,  9, 10, 11, 12]]),
 Tensor(shape=[2, 1], dtype=Int64, value=
 [[ 6],
  [13]]))
```

mindspore.ops.tile

mindspore.ops.**tile**(*input*, *dims*)

Creates a new tensor by repeating the elements in the input tensor *dims* times.

The i'th dimension of output tensor has *input.shape[i] * dims[i]* elements, and the values of *input* are repeated *dims[i]* times along the i'th dimension.

Note:

- On Ascend, the number of *dims* should not exceed 8, and currently does not support scenarios where more than 4 dimensions are repeated simultaneously.
- If *input.dim* = d, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * y_1, x_2 * y_2, ..., x_S * y_S)$.
- If *input.dim* < d, prepend 1 to the shape of *input* until their lengths are consistent. Such as set the shape of *input* as $(1, ..., x_1, x_2, ..., x_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(1 * y_1, ..., x_R * y_R, x_S * y_S)$.
- If *input.dim* > d, prepend 1 to *dims* until their lengths are consistent. Such as set the *dims* as $(1, ..., y_1, y_2, ..., y_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * 1, ..., x_R * y_R, x_S * y_S)$.

Parameters

- **input** (Tensor) – The input tensor.
- **dims** (*tuple[int]*) – The specified number of repetitions in each dimension.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2], [3, 4]])
>>> mindspore.ops.tile(input, (2, 3))
Tensor(shape=[4, 6], dtype=Int64, value=
[[1, 2, 1, 2, 1, 2],
 [3, 4, 3, 4, 3, 4],
 [1, 2, 1, 2, 1, 2],
 [3, 4, 3, 4, 3, 4]])
>>> mindspore.ops.tile(input, (2, 3, 2))
Tensor(shape=[2, 6, 4], dtype=Int64, value=
[[[1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4]],
 [[1, 2, 1, 2],
  [3, 4, 3, 4],
```

(continues on next page)

(continued from previous page)

```
  [1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4]]])
```

mindspore.ops.tril

mindspore.ops.**tril**(*input*, *diagonal=0*)

Zero the input tensor above the diagonal specified.

Parameters

- **input** (Tensor) – The input tensor. The rank must be at least 2.
- **diagonal** (*int, optional*) – The diagonal specified of 2-D tensor. Default 0 represents the main diagonal.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[ 1,  2,  3,  4],
...                           [ 5,  6,  7,  8],
...                           [10, 11, 12, 13],
...                           [14, 15, 16, 17]])
>>> mindspore.ops.tril(input)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  0,  0,  0],
 [ 5,  6,  0,  0],
 [10, 11, 12,  0],
 [14, 15, 16, 17]])
>>> mindspore.ops.tril(input, 1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  0,  0],
 [ 5,  6,  7,  0],
 [10, 11, 12, 13],
 [14, 15, 16, 17]])
>>> mindspore.ops.tril(input, -1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 0,  0,  0,  0],
 [ 5,  0,  0,  0],
 [10, 11,  0,  0],
 [14, 15, 16,  0]])
>>> input = mindspore.tensor([[[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]],
...                           [[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]]])
```

(continues on next page)

(continued from previous page)

```
>>> mindspore.ops.tril(input)
Tensor(shape=[2, 3, 3], dtype=Int64, value=
[[[ 1,  0,  0],
  [ 5,  6,  0],
  [10, 11, 12]],
 [[ 1,  0,  0],
  [ 5,  6,  0],
  [10, 11, 12]]])
```

mindspore.ops.triu

mindspore.ops.**triu**(*input*, *diagonal=0*)

Zero the input tensor below the diagonal specified.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **diagonal** (*int, optional*) –The diagonal specified of 2-D tensor. Default 0 represents the main diagonal.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[ 1,  2,  3,  4],
...                           [ 5,  6,  7,  8],
...                           [10, 11, 12, 13],
...                           [14, 15, 16, 17]])
>>> mindspore.ops.triu(input)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  3,  4],
 [ 0,  6,  7,  8],
 [ 0,  0, 12, 13],
 [ 0,  0,  0, 17]])
>>> mindspore.ops.triu(input, 1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 0,  2,  3,  4],
 [ 0,  0,  7,  8],
 [ 0,  0,  0, 13],
 [ 0,  0,  0,  0]])
>>> mindspore.ops.triu(input, -1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  3,  4],
 [ 5,  6,  7,  8],
```

(continues on next page)

(continued from previous page)

```
 [ 0, 11, 12, 13],
 [ 0,  0, 16, 17]])
>>> input = mindspore.tensor([[[ 1,  2,  3],
...                           [ 5,  6,  7],
...                           [10, 11, 12]],
...                          [[ 1,  2,  3],
...                           [ 5,  6,  7],
...                           [10, 11, 12]]])
>>> mindspore.ops.triu(input)
Tensor(shape=[2, 3, 3], dtype=Int64, value=
[[[ 1,  2,  3],
  [ 0,  6,  7],
  [ 0,  0, 12]],
 [[ 1,  2,  3],
  [ 0,  6,  7],
  [ 0,  0, 12]]])
```

mindspore.ops.transpose

mindspore.ops.**transpose**(*input*, *input_perm*)

Transpose dimensions of the input tensor according to input permutation.

Note: On GPU and CPU, if the value of *input_perm* is negative, its actual value is *input_perm[i] + rank(input)*. Negative value of *input_perm* is not supported on Ascend.

Parameters

- **input** (Tensor) –The input tensor.
- **input_perm** (*tuple[int]*) –Specify the new axis ordering.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]], mindspore.
 ↪float32)
>>> output = mindspore.ops.transpose(input, (0, 2, 1))
>>> print(output)
[[[ 1.  4.]
  [ 2.  5.]
  [ 3.  6.]]
 [[ 7. 10.]
  [ 8. 11.]
  [ 9. 12.]]]
```

mindspore.ops.unbind

mindspore.ops.**unbind**(*input*, *dim=0*)

Remove a tensor dimension, return a tuple of all slices along a given dimension.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*int*) –Specipy the dimension to remove. Default 0 .

Returns

tuple of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> output = mindspore.ops.unbind(x, dim=0)
>>> print(output)
(Tensor(shape=[3], dtype=Int64, value=[1, 2, 3]), Tensor(shape=[3], dtype=Int64, value=[4, 5,
 ↪ 6]),
Tensor(shape=[3], dtype=Int64, value=[7, 8, 9]))
```

mindspore.ops.unique

mindspore.ops.**unique**(*input*)

Remove duplicate elements from the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) –The input tensor.

Returns

Tuple(output, indices) of 2 tensors.

- **output** (Tensor) - The deduplicated output tensor.
- **indices** (Tensor) - The indices of the elements of the input tensor in the *output* .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 5, 2], mindspore.int32)
>>> output, indices = mindspore.ops.unique(x)
>>> print(output)
[1 2 5]
>>> print(indices)
[0 1 2 1]
```

mindspore.ops.unique_consecutive

mindspore.ops.**unique_consecutive**(*input*, *return_inverse=False*, *return_counts=False*, *dim=None*)

Remove consecutive duplicate elements in the input tensor, retaining only the first occurrence from each repeated group.

Parameters

- **input** (Tensor) –The input tensor.
- **return_inverse** (*bool, optional*) –Whether to also return the indices for where elements in the original input ended up in the returned unique list. Default False .
- **return_counts** (*bool, optional*) –Whether to also return the counts for each unique element. Default False .
- **dim** (*int, optional*) –Specify the dimension for unique. Default None , the input tensor will be flattened.

Returns

Tensor or tuple(output, inverse_indices, counts) of tensors.

- **output** (Tensor) - The deduplicated output tensor.
- **inverse_indices** (Tensor, optional) - The indices of the elements of the input tensor in the *output* .
- **counts** (Tensor, optional) - The counts for each unique element.

Supported Platforms:

Ascend GPU CPU

Examples

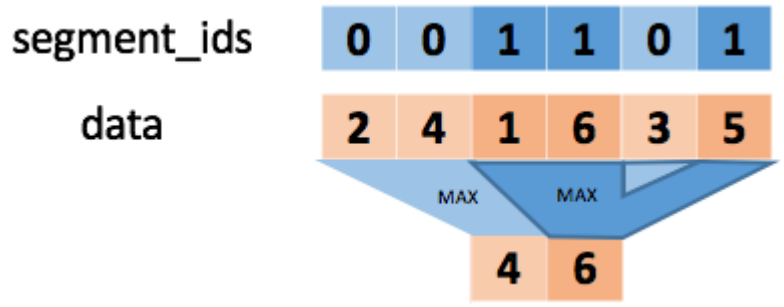
```
>>> import mindspore
>>> x = mindspore.tensor([1, 1, 2, 2, 3, 1, 1, 2], mindspore.int32)
>>> output, inverse_indices, counts = mindspore.ops.unique_consecutive(x, True, True, None)
>>> print(output)
[1 2 3 1 2]
>>> print(inverse_indices)
[0 0 1 1 2 3 3 4]
>>> print(counts)
[2 2 1 2 1]
```

mindspore.ops.unsorted_segment_max

mindspore.ops.**unsorted_segment_max**(*x*, *segment_ids*, *num_segments*)

Compute the maximum of the input tensor along segments.

The following figure shows the calculation process of unsorted_segment_max:



$$\text{output}_{i} = \max_{j...} \text{data}\,[j\ldots]$$

where *max* over tuples $j...$ such that $segment\_ids[j...] == i$.

Note:

- If the segment_id i is absent in the segment_ids, then output[i] will be filled with the minimum value of the x's type.
- The *segment_ids* must be non-negative tensor.

Parameters

- **x** (Tensor) −The input tensor.
- **segment_ids** (Tensor) −Indicate the segment to which each element belongs. Set the shape as $(x_1, x_2, ..., x_N)$, where 0 < N <= R.
- **num_segments** (*Union[int, Tensor]*) −Number of segments, it can be an int or 0-D tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

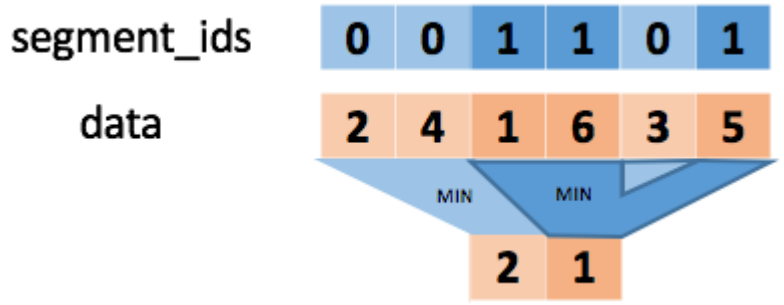
```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [4, 2, 1]])
>>> segment_ids = mindspore.tensor([0, 1, 1])
>>> num_segments = 2
>>> mindspore.ops.unsorted_segment_max(x, segment_ids, num_segments)
Tensor(shape=[2, 3], dtype=Int64, value=
[[1, 2, 3],
 [4, 5, 6]])
```

mindspore.ops.unsorted_segment_min

mindspore.ops.**unsorted_segment_min**(*x*, *segment_ids*, *num_segments*)

Compute the minimum of the input tensor along segments.

The following figure shows the calculation process of unsorted_segment_min:



$$\text{output}_{i} = \min_{j\ldots} \text{data}\,[j\ldots]$$

where *min* over tuples $j\ldots$ such that $segment\_ids[j\ldots] == i$.

Note:

- If the segment_id i is absent in the segment_ids, then output[i] will be filled with the maximum value of the x's type.
- The *segment_ids* must be non-negative tensor.

Parameters

- **x** (Tensor) −The input tensor.
- **segment_ids** (Tensor) −Indicate the segment to which each element belongs.
- **num_segments** (*Union[int, Tensor], optional*) −Number of segments, it can be an int or 0-D tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

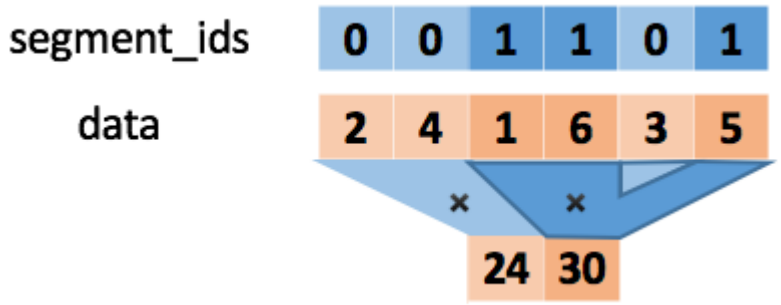
```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [4, 2, 1]])
>>> segment_ids = mindspore.tensor([0, 1, 1])
>>> num_segments = 2
>>> mindspore.ops.unsorted_segment_min(x, segment_ids, num_segments)
Tensor(shape=[2, 3], dtype=Int64, value=
[[1, 2, 3],
 [4, 2, 1]])
```

mindspore.ops.unsorted_segment_prod

mindspore.ops.**unsorted_segment_prod**(*x*, *segment_ids*, *num_segments*)

Compute the product of the input tensor along segments.

The following figure shows the calculation process of unsorted_segment_prod:



Note:

- If the segment_id i is absent in the segment_ids, then output[i] will be filled with 1.
- The *segment_ids* must be non-negative tensor.

Parameters

- **x** (Tensor) −The input tensor.
- **segment_ids** (Tensor) −Indicate the segment to which each element belongs.
- **num_segments** (*Union[int, Tensor], optional*) −Number of segments, it can be an int or 0-D tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

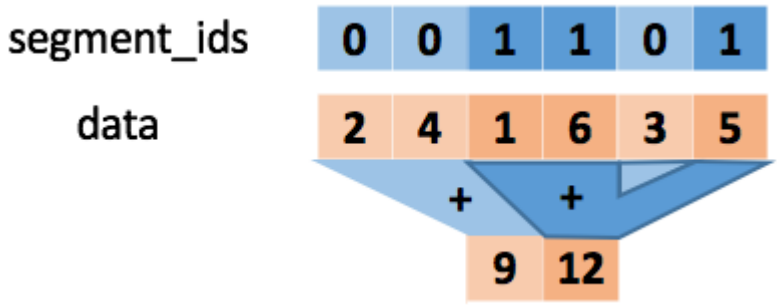
```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3], [4, 5, 6], [4, 2, 1]])
>>> segment_ids = mindspore.tensor([0, 1, 0])
>>> num_segments = 2
>>> mindspore.ops.unsorted_segment_prod(x, segment_ids, num_segments)
Tensor(shape=[2, 3], dtype=Int64, value=
[[4, 4, 3],
 [4, 5, 6]])
```

mindspore.ops.unsorted_segment_sum

mindspore.ops.**unsorted_segment_sum**(*input_x*, *segment_ids*, *num_segments*)

Compute the sum of the input tensor along segments.

The following figure shows the calculation process of unsorted_segment_sum:



$$\text{output}[i] = \sum_{segment\_ids[j]==i} \text{data}[j, \ldots]$$

where j, ... is a tuple describing the index of element in data. *segment_ids* selects which elements in data to sum up. Segment_ids does not need to be sorted, and it does not need to cover all values in the entire valid value range.

Note:

- If the segment_id i is absent in the segment_ids, then output[i] will be filled with 0.
- On Ascend, if the value of segment_id is less than 0 or greater than the length of the input data shape, an execution error will occur.

If the sum of the given segment_ids i is empty, then $\text{output}[i] = 0$. If the given segment_ids is negative, the value will be ignored. 'num_segments' must be equal to the number of different segment_ids.

Parameters

- **input_x** (Tensor) –The input tensor.
- **segment_ids** (Tensor) –Indicate the segment to which each element belongs.
- **num_segments** (*Union[int, Tensor], optional*) –Number of segments, it can be an int or 0-D tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([1, 2, 3, 4])
>>> segment_ids = mindspore.tensor([0, 0, 1, 2])
>>> num_segments = 3
>>> mindspore.ops.unsorted_segment_sum(input_x, segment_ids, num_segments)
Tensor(shape=[3], dtype=Int64, value= [3, 3, 4])
>>> input_x = mindspore.tensor([1, 2, 3, 4, 2, 5])
>>> segment_ids = mindspore.tensor([0, 0, 1, 2, 3, 4])
>>> num_segments = 6
>>> mindspore.ops.unsorted_segment_sum(input_x, segment_ids, num_segments)
Tensor(shape=[6], dtype=Int64, value= [3, 3, 4, 2, 5, 0])
```

mindspore.ops.unsqueeze

mindspore.ops.**unsqueeze**(*input*, *dim*)

Adds an additional dimension to the input tensor at the given dimension.

Parameters

- **input** (Tensor) −The input tensor.
- **dim** (*int*) −The dimension specified.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3, 4])
>>> mindspore.ops.unsqueeze(input, 0)
Tensor(shape=[1, 4], dtype=Int64, value=
[[1, 2, 3, 4]])
>>> mindspore.ops.unsqueeze(input, 1)
Tensor(shape=[4, 1], dtype=Int64, value=
[[1],
 [2],
 [3],
 [4]])
```

mindspore.ops.unstack

mindspore.ops.**unstack**(*input_x*, *axis=0*)

Unstack the input tensor along the specified axis.

Parameters

- **input_x** (Tensor) −The input tensor.
- **axis** (*int*) −The specified axis. Default 0 .

Returns

Tuple of tensors

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 1, 1, 1], [2, 2, 2, 2]])
>>> mindspore.ops.unstack(input, 0)
(Tensor(shape=[4], dtype=Int64, value= [1, 1, 1, 1]),
 Tensor(shape=[4], dtype=Int64, value= [2, 2, 2, 2]))
```

mindspore.ops.view_as_real

mindspore.ops.**view_as_real**(*input*)

Return a real tensor with the last dimension of size 2, composed of the real and imaginary parts of the complex elements in the input tensor.

Parameters

input (Tensor) −The complex input tensor.

Returns

A real tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([2+1j,2+3j,2-1j,2])
>>> output = mindspore.ops.view_as_real(input)
>>> print(output)
[[ 2.  1.]
 [ 2.  3.]
 [ 2. -1.]
 [ 2.  0.]]
```

mindspore.ops.vsplit

mindspore.ops.**vsplit**(*input*, *indices_or_sections*)

Split the input tensor with two or more dimensions, into multiple sub-tensors vertically according to *indices_or_sections*.

It is equivalent to *ops.tensor_split* with $axis = 0$.

Parameters

- **input** (Tensor) –The input tensor.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –See *indices_or_sections* argument in *mindspore.ops.tensor_split()*.

Returns

Tuple of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[ 0,  1,  2,  3],
...                           [ 4,  5,  6,  7],
...                           [ 8,  9, 10, 11],
...                           [12, 13, 14, 15]])
>>> mindspore.ops.vsplit(input, 2)
(Tensor(shape=[2, 4], dtype=Int64, value=
 [[0, 1, 2, 3],
  [4, 5, 6, 7]]),
 Tensor(shape=[2, 4], dtype=Int64, value=
 [[ 8,  9, 10, 11],
  [12, 13, 14, 15]]))
>>> mindspore.ops.vsplit(input, [3, 6])
(Tensor(shape=[3, 4], dtype=Int64, value=
 [[ 0,  1,  2,  3],
  [ 4,  5,  6,  7],
  [ 8,  9, 10, 11]]),
 Tensor(shape=[1, 4], dtype=Int64, value=
 [[12, 13, 14, 15]]),
 Tensor(shape=[0, 4], dtype=Int64, value=
 ))
```

mindspore.ops.vstack

mindspore.ops.**vstack**(*inputs*)

Stacks tensors in sequence vertically.

This is equivalent to concatenation along the first axis. 1-D tensors $(N,)$ should firstly be reshaped to $(1, N)$, and then be concatenated along the first axis.

Parameters

inputs (*Union(List[tensor], Tuple[tensor])*) –The 1-D or 2-D input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1, 2, 3])
>>> x2 = mindspore.tensor([4, 5, 6])
>>> mindspore.ops.vstack((x1, x2))
Tensor(shape=[2, 3], dtype=Int64, value=
[[1, 2, 3],
 [4, 5, 6]])
>>> x1 = mindspore.tensor([[1],[2],[3]])
>>> x2 = mindspore.tensor([[4],[5],[6]])
>>> mindspore.ops.vstack([x1, x2])
Tensor(shape=[6, 1], dtype=Int64, value=
[[1],
 [2],
 [3],
 [4],
 [5],
 [6]])
```

mindspore.ops.where

`mindspore.ops.`**`where`**(*condition*, *input*, *other*)

Return a tensor in which the elements are selected from *input* or *other* based on the *condition*.

Support broadcasting.

$$output_i = \begin{cases} input_i, & if\ condition_i \\ other_i, & otherwise \end{cases}$$

Parameters

- **`condition`** (`Tensor[bool]`) −If True, yield *input*, otherwise yield *other*.
- **`input`** (`Union[Tensor, Scalar]`) −When *condition* is True, values to select from.
- **`other`** (`Union[Tensor, Scalar]`) −When *condition* is False, values to select from.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[0, 1],
...                           [2, 3]])
>>> other = mindspore.tensor([[1, 1],
...                           [1, 1]])
>>> condition = input < 3
>>> mindspore.ops.where(condition, input, other)
Tensor(shape=[2, 2], dtype=Int64, value=
[[0, 1],
 [2, 1]])
```

mindspore.ops.cross

mindspore.ops.**cross**(*input*, *other*, *dim=None*)

Compute the cross product of two input tensors along the specified dimension.

Note: *input* and *other* must have the same shape, and the size of their *dim* dimension should be *3*. If *dim* is not specified, it is set to be the first dimension found with the size *3*.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.
- **dim** (*int, optional*) –Specify the dimension for computation. Default None.

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
>>> other = mindspore.tensor([[4, 5, 6], [4, 5, 6], [4, 5, 6]])
>>> mindspore.ops.cross(input, other)
Tensor(shape=[3, 3], dtype=Int64, value=
[[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]])
>>> mindspore.ops.cross(input, other, dim=1)
Tensor(shape=[3, 3], dtype=Int64, value=
[[-3,  6, -3],
 [-3,  6, -3],
 [-3,  6, -3]])
```

mindspore.ops.renorm

mindspore.ops.**renorm**(*input*, *p*, *axis*, *maxnorm*)

Returns a tensor where each subtensor along the specified dimension is renormalized such that its *p* norm is less than or equal to *maxnorm*. If the *p* norm exceeds *maxnorm*, return the values that are obtained by dividing the original values of the subtensor by its *p* norm and then multiplying by *maxnorm*.

Parameters

- **input** (Tensor) –The input tensor.
- **p** (*int*) –The power of norm calculation.
- **axis** (*int*) –Specify the axis for computation.
- **maxnorm** (*float32*) –The max norm specified.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 1, 1], [2, 2, 2], [3, 3, 3]])
>>> output = mindspore.ops.norm(input, 1, 1)
>>> print(output)
[3. 6. 9.]
>>> output = mindspore.ops.renorm(input, 1, 0, 6.)
>>> print(output)
[[1. 1. 1.]
 [2. 2. 2.]
 [2. 2. 2.]]
```

2.1.4 Type Cast

API Name	Description	Supported Platforms
mindspore.ops.cast	Returns a tensor with the new specified data type.	Ascend GPU CPU
mindspore.ops.is_tensor	Check whether the input object is *mindspore.Tensor*.	Ascend GPU CPU
mindspore.ops.scalar_to_tensor	Converts a scalar to a tensor with the specified dtype.	Ascend GPU CPU
mindspore.ops.tuple_to_array	Converts a tuple to a tensor.	Ascend GPU CPU

mindspore.ops.cast

mindspore.ops.**cast**(*input*, *dtype*)

Returns a tensor with the new specified data type.

Note: When converting complex numbers to boolean type, the imaginary part of the complex number is not taken into account. As long as the real part is non-zero, it returns `True`; otherwise, it returns `False`.

Parameters

- **input** (*Union[Tensor, Number]*) – The input tensor or number.
- **dtype** (*dtype.Number*) – The dtype after conversion. Only constant value is allowed.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> dtype = mindspore.float64
>>> output = mindspore.ops.cast(input, dtype)
>>> print(output.dtype)
Float64
>>> print(output)
[1. 2. 3.]
```

mindspore.ops.is_tensor

mindspore.ops.**is_tensor**(*obj*)

Check whether the input object is *mindspore.Tensor*.

Parameters

obj (*Object*) – input object.

Returns

Bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([1.9, 2.2, 3.1])
>>> mindspore.ops.is_tensor(a)
True
```

mindspore.ops.scalar_to_tensor

mindspore.ops.**scalar_to_tensor**(*input_x*, *dtype=mstype.float32*)

Converts a scalar to a tensor with the specified dtype.

Parameters

- **input_x** (*Union[bool, int, float]*) – The input scalar. Only constant value is allowed.
- **dtype** (mindspore.dtype) – The dtype of returned tensor. Only constant value is allowed.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> data = 1
>>> output = mindspore.ops.scalar_to_tensor(data, mindspore.float32)
>>> print(output)
1.0
```

mindspore.ops.tuple_to_array

mindspore.ops.**tuple_to_array**(*input_x*)

Converts a tuple to a tensor.

Note: If the type of the first number in the tuple is integer, the data type of the output tensor is int. Otherwise, the data type of the output tensor is float.

Parameters

input_x (*tuple*) – A tuple of numbers. Only constant value is allowed.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = (1, 2, 3)
>>> output = mindspore.ops.tuple_to_array(input_x)
>>> print(type(output))
<class 'mindspore.common.tensor.Tensor'>
>>> print(output)
[1 2 3]
```

2.1.5 Gradient Clipping

API Name	Description	Supported Platforms
mindspore.ops.clip_by_global_norm	Clips tensor values by the ratio of the sum of their norms.	Ascend GPU CPU
mindspore.ops.clip_by_value	Clips tensor values to a specified min and max.	Ascend GPU CPU
mindspore.ops.clip_by_norm	The input Tensor is cropped based on norm.	Ascend GPU CPU

mindspore.ops.clip_by_global_norm

mindspore.ops.**clip_by_global_norm**(*x*, *clip_norm=1.0*, *use_norm=None*)

Clips tensor values by the ratio of the sum of their norms.

Note:

- On the SEMI_AUTO_PARALLEL mode or AUTO_PARALLEL mode, if the input *x* is the gradient, the gradient norm values on all devices will be automatically aggregated by allreduce inserted after the local square sum of the gradients.

Parameters

- **x** (*Union(tuple[Tensor], list[Tensor])*) – Input data to clip.
- **clip_norm** (*Union(float, int)*) – The clipping ratio, it should be greater than 0. Default `1.0` .
- **use_norm** (*None*) – The global norm. Currently only none is supported. Default `None` .

Returns

Tuple of tensors

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([[2., 3.], [1., 2.]], dtype=mindspore.float32)
>>> x2 = mindspore.tensor([[1., 4.], [3., 1.]], dtype=mindspore.float32)
>>> input_x = (x1, x2)
>>> out = mindspore.ops.clip_by_global_norm(input_x, 1.0)
>>> print(out)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 2.98142403e-01,  4.47213590e-01],
 [ 1.49071202e-01,  2.98142403e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 1.49071202e-01,  5.96284807e-01],
 [ 4.47213590e-01,  1.49071202e-01]]))
```

mindspore.ops.clip_by_value

mindspore.ops.**clip_by_value**(*x*, *clip_value_min=None*, *clip_value_max=None*)

Clips tensor values to a specified min and max.

Limits the value of x to a range, whose lower limit is *clip_value_min* and upper limit is *clip_value_max* .

$$out_i = \begin{cases} clip\_value\_max & \text{if } x_i \geq clip\_value\_max \\ x_i & \text{if } clip\_value\_min < x_i < clip\_value\_max \\ clip\_value\_min & \text{if } x_i \leq clip\_value\_min \end{cases}$$

Note:

- *clip_value_min* and *clip_value_max* cannot be None at the same time;
- When *clip_value_min* is None and *clip_value_max* is not None, the elements in Tensor larger than *clip_value_max* will become *clip_value_max*;
- When *clip_value_min* is not None and *clip_value_max* is None, the elements in Tensor smaller than *clip_value_min* will become *clip_value_min*;
- If *clip_value_min* is greater than *clip_value_max*, the value of all elements in Tensor will be set to *clip_value_max*;
- The data type of *x*, *clip_value_min* and *clip_value_max* should support implicit type conversion and cannot be bool type.

Parameters

- **x** (*Union(Tensor, list[Tensor], tuple[Tensor])*) – Input data, which type is Tensor or a list or tuple of Tensor. Tensors of arbitrary dimensions are supported.
- **clip_value_min** (*Union(Tensor, float, int)*) – The minimum value. Default: `None` .
- **clip_value_max** (*Union(Tensor, float, int)*) – The maximum value. Default: `None` .

Returns

(Union(Tensor, tuple[Tensor], list[Tensor])), a clipped Tensor or a tuple or a list of clipped Tensor. The data type and shape are the same as x.

Raises

- **ValueError** – If both *clip_value_min* and *clip_value_max* are None.
- **TypeError** – If the type of *x* is not in Tensor or list[Tensor] or tuple[Tensor].

- **TypeError** –If the type of *clip_value_min* is not in None, Tensor, float or int.
- **TypeError** –If the type of *clip_value_max* is not in None, Tensor, float or int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> # case 1: the data type of x is Tensor
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> min_value = Tensor(5, mindspore.float32)
>>> max_value = Tensor(20, mindspore.float32)
>>> x = Tensor(np.array([[1., 25., 5., 7.], [4., 11., 6., 21.]]), mindspore.float32)
>>> output = ops.clip_by_value(x, min_value, max_value)
>>> print(output)
[[ 5. 20.  5.  7.]
 [ 5. 11.  6. 20.]]
>>> # case 2: the data type of x is list[Tensor]
>>> min_value = 5
>>> max_value = 20
>>> x = Tensor(np.array([[1., 25., 5., 7.], [4., 11., 6., 21.]]), mindspore.float32)
>>> y = Tensor(np.array([[1., 25., 5., 7.], [4., 11., 6., 21.]]), mindspore.float32)
>>> output = ops.clip_by_value([x,y], min_value, max_value)
>>> for out in output:
...     print(out)
[[ 5. 20.  5.  7.]
 [ 5. 11.  6. 20.]]
[[ 5. 20.  5.  7.]
 [ 5. 11.  6. 20.]]
```

mindspore.ops.clip_by_norm

`mindspore.ops.`**`clip_by_norm`**(*x*, *max_norm*, *norm_type=2.0*, *error_if_nonfinite=False*)

The input Tensor is cropped based on norm. The computation is done by concatenating the norms of all the input elementsinto a vector and then computing the norm of that vector. The Tensor gradient value corresponding to the *identifier*.

Note: The interface is suitable for gradient clipping scenarios, and only supports input of type float.

Parameters

- **x** (*Union[Tensor, list[Tensor], tuple[Tensor]]*) –Input that wishes to be clipped.
- **max_norm** (*Union[float, int]*) –The upper limit of the norm for this group of network parameters.
- **norm_type** (*Union[float, int], optional*) –Norm type. Default: `2.0`.
- **error_if_nonfinite** (*bool, optional*) –If it is `True`, an exception is thrown if the total norm from the input is nan, inf or -inf. If it is `False`, no exception will be thrown.Default: `False` .

Returns

Tensors, a list or tuple of Tensors, representing clipped Tensors.

Raises

RuntimeError – If the total norm from the *x* is nan, inf or -inf.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> x = Tensor([[0.8748, 0.1425, 0.0076], [0.7721, 0.4084, 0.0552], [4.6376, 0.2914, 2.
↪1120]])
>>> out = ops.clip_by_norm(x, max_norm=1)
>>> print(out)
[[0.16650201 0.02712224 0.00144652]
 [0.14695495 0.07773139 0.0105063 ]
 [0.8826814  0.0554626  0.40198016]]
```

2.2 Mathematical Functions

2.2.1 Element-wise Operations

API Name	Description	Supported Platforms
mindspore.ops.abs	Compute the absolute value of a tensor element-wise.	`Ascend` `GPU` `CPU`
mindspore.ops.absolute	Alias for *mindspore.ops.abs()* .	`Ascend` `GPU` `CPU`
mindspore.ops.accumulate_n	Return the element-wise sum of all input tensors.	`Ascend` `GPU`
mindspore.ops.acos	Compute arccosine of each element in input tensors.	`Ascend` `GPU` `CPU`
mindspore.ops.arccos	Alias for *mindspore.ops.acos()* .	`Ascend` `GPU` `CPU`
mindspore.ops.acosh	Computes inverse hyperbolic cosine of each element in inputs tensors.	`Ascend` `GPU` `CPU`
mindspore.ops.add	Compute the element-wise sum of the two input tensors.	`Ascend` `GPU` `CPU`
mindspore.ops.addcdiv	Divide *tensor1* by *tensor2* element-wise, multiply the result by the scalar *value* , and add it to *input* .	`Ascend` `GPU` `CPU`
mindspore.ops.addcmul	Multiply *tensor1* by *tensor2* element-wise, scale the result by the scalar *value* , and add it to *input* .	`Ascend` `GPU` `CPU`
mindspore.ops.addmv	Multiply the matrix *mat* and vector *vec* , and then add the result to the *input* .	`Ascend` `GPU` `CPU`
mindspore.ops.addn	Return the element-wise sum of all input tensors.	`Ascend` `GPU` `CPU`
mindspore.ops.angle	Returns the element-wise angle of the given complex tensor.	Ascend`` `GPU` `CPU`
mindspore.ops.arccosh	Alias for *mindspore.ops.acosh()*.	`Ascend` `GPU` `CPU`
mindspore.ops.arcsin	Alias for *mindspore.ops.asin()*.	`Ascend` `GPU` `CPU`
mindspore.ops.arcsinh	Alias for *mindspore.ops.asinh()*.	`Ascend` `GPU` `CPU`
mindspore.ops.arctan	Alias for *mindspore.ops.atan()*.	`Ascend` `GPU` `CPU`
mindspore.ops.arctanh	Alias for *mindspore.ops.atanh()*.	`Ascend` `GPU` `CPU`

continues on next page

Table 6 – continued from previous page

mindspore.ops.arctan2	Alias for mindspore.ops.atan2().	Ascend GPU CPU
mindspore.ops.asin	Computes arcsine of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.asinh	Computes inverse hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.atan	Computes the trigonometric inverse tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.atan2	Returns arctangent of input/other element-wise.	Ascend GPU CPU
mindspore.ops.atanh	Computes inverse hyperbolic tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.atleast_1d	Returns a one-dimensional tensor of each zero-dimensional tensor, while tensors with one or more dimensions remain unchanged.	Ascend GPU CPU
mindspore.ops.atleast_2d	Returns a 2-dimensional tensor of each tensor, while tensors with two or more dimensions remain unchanged.	Ascend GPU CPU
mindspore.ops.atleast_3d	Returns a 3-dimensional tensor of each tensor, while tensors with three or more dimensions remain unchanged.	Ascend GPU CPU
mindspore.ops.bessel_i0	Computes the zeroth order modified Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_i0e	Computes the exponentially scaled zeroth order modified Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_i1	Computes the first order modified Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_i1e	Computes the exponentially scaled first order modified Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_j0	Computes the zeroth order Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_j1	Computes the first order Bessel function of the first kind for each element input.	GPU CPU
mindspore.ops.bessel_k0	Computes the zeroth order modified Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bessel_k0e	Computes the exponentially scaled zeroth order modified Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bessel_k1	Computes the first order modified Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bessel_k1e	Computes the exponentially scaled first order modified Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bessel_y0	Computes the zeroth order Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bessel_y1	Computes the first order Bessel function of the second kind for each element input.	GPU CPU
mindspore.ops.bitwise_and	Compute the bitwise AND of two input tensors.	Ascend GPU CPU
mindspore.ops.bitwise_left_shift	Perform a left bitwise shift operation on the *input* element-wise, where the number of bits to shift is specified by *other*.	Ascend GPU CPU
mindspore.ops.bitwise_or	Compute the bitwise OR of two input tensors.	Ascend GPU CPU

continues on next page

Table 6 – continued from previous page

mindspore.ops.bitwise_right_shift	Perform a right bitwise shift operation on the *input* element-wise, where the number of bits to shift is specified by *other*.	Ascend GPU CPU
mindspore.ops.bitwise_xor	Compute the bitwise XOR of two input tensors.	Ascend GPU CPU
mindspore.ops.ceil	Rounds a tensor up to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.clamp	Clamp all elements of the input tensor within the range [min, max].	Ascend GPU CPU
mindspore.ops.clip	Alias for mindspore.ops.clamp() .	Ascend GPU CPU
mindspore.ops.combinations	Return all r-length subsequences of input tensor.	Ascend GPU CPU
mindspore.ops.copysign	Create a float tensor composed of the absolute values of *x* and the signs of *other* .	Ascend GPU CPU
mindspore.ops.cos	Computes cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.cosh	Computes hyperbolic cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.cosine_similarity	Calculate cosine similarity between two input tensors along the specified dimension.	Ascend GPU CPU
mindspore.ops.cov	Return the covariance matrix of the input tensor, where the rows of the input tensor represent variables and the columns represent observations.	Ascend GPU CPU
mindspore.ops.diff	Computes the n-th forward difference along the given axis.	Ascend GPU CPU
mindspore.ops.deg2rad	Convert angles from degrees to radians element-wise.	Ascend GPU CPU
mindspore.ops.digamma	Computes the logarithmic derivative of the gamma function on input tensor.	GPU CPU
mindspore.ops.div	Divide the first input tensor by the second input tensor in floating-point type element-wise.	Ascend GPU CPU
mindspore.ops.divide	Alias for mindspore.ops.div() .	Ascend GPU CPU
mindspore.ops.erf	Compute the Gauss error of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.erfc	Compute the complementary error function of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.erfinv	Compute the inverse error of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.exp	Compute exponential of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.exp2	Compute base two exponential of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.expm1	Compute exponential of the input tensor, then minus 1, element-wise.	Ascend`` GPU CPU
mindspore.ops.floor	Rounds a tensor down to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.floor_div	Alias for mindspore.ops.floor_divide() .	Ascend GPU CPU
mindspore.ops.floor_divide	Compute element-wise division of *input* by *other* and floor the result.	Ascend GPU CPU
mindspore.ops.floor_mod	Compute the remainder of element-wise flooring division of first input by second input.	Ascend GPU CPU
mindspore.ops.float_power	Computes the first input to the power of the second input.	GPU CPU

continues on next page

Table 6 – continued from previous page

mindspore.ops.fmax	Compute the maximum of input tensors element-wise.	CPU
mindspore.ops.fmod	Compute the remainder of element-wise division of first input by the second input.	Ascend GPU CPU
mindspore.ops.frac	Return the fractional part of each element in the input tensor.	Ascend GPU CPU
mindspore.ops.gcd	Computes greatest common divisor of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.hypot	Given the legs of a right triangle, return its hypotenuse, element-wise.	Ascend GPU CPU
mindspore.ops.igamma	Calculates lower regularized incomplete Gamma function.	Ascend GPU CPU
mindspore.ops.igammac	Calculates upper regularized incomplete Gamma function.	Ascend GPU CPU
mindspore.ops.imag	Return a new tensor containing imaginary value of the input tensor, element-wise.	Ascend GPU CPU
mindspore.ops.i0	For details, please refer to mindspore.ops.bessel_i0().	GPU CPU
mindspore.ops.inv	Computes Reciprocal of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.invert	Flip all bits of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.lcm	Computes least common multiplier of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.ldexp	Multiplies input tensor by 2^{other} element-wise.	Ascend GPU CPU
mindspore.ops.lerp	Does a linear interpolation of two tensors input and end based on a float or tensor weight.	Ascend GPU CPU
mindspore.ops.log	Compute the natural logarithm of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.log2	Compute the logarithm to the base 2 of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.log10	Compute the logarithm to the base 10 of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.log1p	Compute the natural logarithm of (tensor + 1) element-wise.	Ascend GPU CPU
mindspore.ops.logaddexp	Computes the logarithm of the sum of exponentiations of the inputs.	Ascend GPU CPU
mindspore.ops.logaddexp2	Computes the logarithm of the sum of exponentiations in base of 2 of the inputs.	Ascend GPU CPU
mindspore.ops.logical_and	Compute the "logical AND" of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.logical_not	Compute the "logical NOT" of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.logical_or	Compute the "logical OR" of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.logical_xor	Compute the "logical XOR" of two tensors element-wise.	Ascend CPU
mindspore.ops.logit	Calculate the logit of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.mul	Multiplies two tensors element-wise.	Ascend GPU CPU
mindspore.ops.multiply	Alias for mindspore.ops.asinh().	Ascend GPU CPU
mindspore.ops.mvlgamma	Compute the multivariate log-gamma function with dimension p element-wise.	Ascend GPU CPU

continues on next page

Table 6 – continued from previous page

mindspore.ops.neg	Returns a tensor with negative values of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.negative	Alias for mindspore.ops.neg() .	Ascend GPU CPU
mindspore.ops.nextafter	Returns the next representable floating-point value after *input* towards *other* element-wise.	Ascend GPU CPU
mindspore.ops.polar	Converts polar coordinates to Cartesian coordinates.	Ascend GPU CPU
mindspore.ops.polygamma	Computes the *n*-th derivative of the polygamma function on *input*.	GPU CPU
mindspore.ops.positive	Return self tensor.	Ascend GPU CPU
mindspore.ops.pow	Calculate the *exponent* power of each element in *input*.	Ascend GPU CPU
mindspore.ops.rad2deg	Converts angles in radians to angles in degrees element-wise.	Ascend GPU CPU
mindspore.ops.ravel	Expand the multidimensional Tensor into 1D along the 0 axis direction.	Ascend GPU CPU
mindspore.ops.real	Return a tensor that is the real part of the input.	Ascend GPU CPU
mindspore.ops.reciprocal	Returns reciprocal of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.remainder	Compute the remainder of division for the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.rot90	Rotate a n-D tensor by 90 degrees in the plane specified by dims axis.	Ascend GPU CPU
mindspore.ops.round	Round elements of input to the nearest integer.	Ascend GPU CPU
mindspore.ops.rsqrt	Compute reciprocal of square root of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.sgn	Extension of mindspore.ops.sign() in complex domain.	Ascend GPU CPU
mindspore.ops.sign	Return an element-wise indication of the sign of a number.	Ascend GPU CPU
mindspore.ops.signbit	Determine the symbol of each element.	Ascend GPU CPU
mindspore.ops.sin	Compute sine of the input tensor element-wise.	Ascend GPU CPU
mindspore.ops.sinc	Compute the normalized sinc of input.	Ascend GPU CPU
mindspore.ops.sinh	Compute hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.sqrt	Return sqrt of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.square	Return square of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.sub	Subtract the second input from the first input element-wise.	Ascend GPU CPU
mindspore.ops.subtract	Subtract *other* scaled by *alpha* from *input*.	Ascend GPU CPU
mindspore.ops.t	Transpose a 2-D tensor.	Ascend GPU CPU
mindspore.ops.tan	Computes tangent of *input* element-wise.	Ascend GPU CPU
mindspore.ops.tanhshrink	Apply the element-wise Tanhshrink function.	Ascend GPU CPU
mindspore.ops.trapz	Compute the trapezoidal rule along dim.	Ascend GPU CPU
mindspore.ops.tril_indices	Return a 2-by-N tensor containing the indices of the lower triangular elements of a *row* * *col* matrix.	Ascend GPU CPU
mindspore.ops.triu_indices	Return a 2-by-N tensor containing the indices of the upper triangular elements of a *row* * *col* matrix.	Ascend GPU CPU
mindspore.ops.true_divide	Alias for mindspore.ops.div() with rounding_mode=None .	Ascend GPU CPU
mindspore.ops.trunc	Returns a tensor with the truncated integer values of the elements of the input tensor.	Ascend GPU CPU

continues on next page

Table 6 – continued from previous page

mindspore.ops.truncate_div	Divide the first input tensor x by the second input tensor y element-wise and rounds the results of division towards zero.	Ascend GPU CPU
mindspore.ops.truncate_mod	Return the remainder of division element-wise.	Ascend GPU CPU
mindspore.ops.xdivy	Divide x by y element-wise.	Ascend GPU CPU
mindspore.ops.xlogy	Compute *input* multiplied by the logarithm of *other* element-wise.	Ascend GPU CPU
mindspore.ops.zeta	Elemental-wise compute the Hurwitz zeta function values.	Ascend GPU CPU

mindspore.ops.abs

mindspore.ops.**abs**(*input*)

Compute the absolute value of a tensor element-wise.

$$out_i = |input_i|$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.abs(mindspore.tensor([-1.0, 1.0, 0.0]))
>>> print(output)
[1. 1. 0.]
```

mindspore.ops.absolute

mindspore.ops.**absolute**(*input*)

Alias for *mindspore.ops.abs()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.accumulate_n

mindspore.ops.**accumulate_n**(*x*)

Return the element-wise sum of all input tensors.

mindspore.ops.accumulate_n() is similar to *mindspore.ops.addn()*, but accumulate_n will not wait for all of its inputs to be ready before summing, which is able to reduce peak memory.

Parameters

x (*Union(tuple[Tensor], list[Tensor])*) – List of tensors or tuple of tensors.

Returns

Tensor

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> y = mindspore.tensor([4, 5, 6], mindspore.float32)
>>> output = mindspore.ops.accumulate_n([x, y, x, y])
>>> print(output)
[10. 14. 18.]
```

mindspore.ops.acos

mindspore.ops.**acos**(*input*)

Compute arccosine of each element in input tensors.

$$out_i = \cos^{-1}(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.acos(mindspore.tensor([0.74, 0.04, 0.30, 0.56]))
>>> print(output)
[0.737726  1.5307857 1.2661036 0.9764105]
```

mindspore.ops.arccos

mindspore.ops.**arccos**(*input*)

Alias for *mindspore.ops.acos()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.acosh

mindspore.ops.**acosh**(*input*)

Computes inverse hyperbolic cosine of each element in inputs tensors.

$$out_i = \cosh^{-1}(input_i)$$

Note: Given an input tensor input, the function computes inverse hyperbolic cosine of every element. Input range is [1, inf].

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.acosh(mindspore.tensor([1.0, 1.5, 3.0, 100.0]))
>>> print(output)
[0.        0.9624237 1.7627472 5.298292 ]
```

mindspore.ops.add

mindspore.ops.**add**(*input*, *other*)

Compute the element-wise sum of the two input tensors.

$$out_i = input_i + other_i$$

Note:

- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- Support broadcast, support implicit type conversion and type promotion.
- When the input is a tensor, the dimension should be greater than or equal to 1.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – The first input tensor.
- **other** (*Union[Tensor, number.Number, bool]*) – The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: x and y are both tensor.
>>> x = mindspore.tensor([1., 2., 3.])
>>> y = mindspore.tensor([4., 5., 6.])
>>> output = mindspore.ops.add(x, y)
>>> print(output)
[5. 7. 9.]
>>> # case 2: x is a scalar and y is a tensor
>>> x = mindspore.tensor(1, mindspore.int32)
>>> y = mindspore.tensor([4., 5., 6.])
>>> output = mindspore.ops.add(x, y)
>>> print(output)
[5. 6. 7.]
>>> # the data type of x is int32, the data type of y is float32,
>>> # and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

mindspore.ops.addcdiv

mindspore.ops.**addcdiv**(*input*, *tensor1*, *tensor2*, *value=1*)

Divide *tensor1* by *tensor2* element-wise, multiply the result by the scalar *value* , and add it to *input* .

$$y[i] = input[i] + value[i] * (tensor1[i]/tensor2[i])$$

Parameters

- **input** (Tensor) –The input tensor.
- **tensor1** (Tensor) –Tensor1, the numerator.
- **tensor2** (Tensor) –Tensor2, the denominator.
- **value** (*Union[Tensor, number]*) –The multiplier for (*tensor1* / *tensor2*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 1, 1, 1], mindspore.float32)
>>> x1 = mindspore.tensor([1, 2, 3, 4], mindspore.float32)
>>> x2 = mindspore.tensor([4, 4, 2, 1], mindspore.float32)
>>> y = mindspore.ops.addcdiv(x, x1, x2, 0.1)
>>> print(y)
[1.025 1.05  1.15  1.4  ]
```

mindspore.ops.addcmul

mindspore.ops.**addcmul**(*input*, *tensor1*, *tensor2*, *value=1*)

Multiply *tensor1* by *tensor2* element-wise, scale the result by the scalar *value* , and add it to *input* .

$$output[i] = input[i] + value[i] * (tensor1[i] * tensor2[i])$$

Parameters

- **input** (Tensor) –The input tensor.
- **tensor1** (Tensor) –The first tensor to be multiplied.
- **tensor2** (Tensor) –The second tensor to be multiplied.
- **value** (*Union[Tensor, number]*) –The multiplier for (*tensor1* * *tensor2*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor(([1, 1, 1]), mindspore.float32)
>>> x1 = mindspore.tensor([[1], [2], [3]], mindspore.float32)
>>> x2 = mindspore.tensor([[1, 2, 3]], mindspore.float32)
>>> value = mindspore.tensor([1], mindspore.float32)
>>> y = mindspore.ops.addcmul(x, x1, x2, value)
>>> print(y)
[[ 2.  3.  4.]
 [ 3.  5.  7.]
 [ 4.  7. 10.]]
```

mindspore.ops.addmv

mindspore.ops.**addmv**(*input*, *mat*, *vec*, *, *beta=1*, *alpha=1*)

Multiply the matrix *mat* and vector *vec* , and then add the result to the *input* .

Note:

- If mat is a (N, M) tensor, vec is a 1-D tensor of size M, then *input* must be broadcastable with a 1-D tensor of size N, *out* will be a 1-D tensor of size N.
- If *beta* is 0, *input* will be ignored.

$$output = input + (mat@vec)$$

Parameters

- **input** (Tensor) –The input tensor.
- **mat** (Tensor) –The matrix tensor to be multiplied.
- **vec** (Tensor) –The vector tensor to be multiplied.

Keyword Arguments

- **beta** (*scalar[int, float, bool], optional*) –Scale factor for *input* . Default 1 .
- **alpha** (*scalar[int, float, bool], optional*) –Scale factor for (*mat* @ *vec*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([2., 3.])
>>> mat = mindspore.tensor([[2., 5., 3.], [4., 2., 2.]])
>>> vec = mindspore.tensor([3., 2., 4.])
>>> output = mindspore.ops.addmv(input, mat, vec)
>>> print(output)
[30. 27.]
```

mindspore.ops.addn

mindspore.ops.**addn**(*x*)

Return the element-wise sum of all input tensors.

Parameters

x (*Union(tuple[Tensor], list[Tensor])*) – List of tensors or tuple of tensors.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3])
>>> y = mindspore.tensor([4, 5, 6])
>>> mindspore.ops.addn([x, y, x, y])
Tensor(shape=[3], dtype=Int64, value= [10, 14, 18])
```

mindspore.ops.angle

mindspore.ops.**angle**(*input*)

Returns the element-wise angle of the given complex tensor.

$$output_i = angle(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1 + 1j, -2 + 2j, 3 - 3j], mindspore.complex64)
>>> output = mindspore.ops.angle(input)*180/3.14159
>>> print(output)
[135.0001  135.0001  -45.00004]
```

mindspore.ops.arccosh

mindspore.ops.**arccosh**(*input*)

Alias for *mindspore.ops.acosh()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.arcsin

mindspore.ops.**arcsin**(*x*)

Alias for *mindspore.ops.asin()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.arcsinh

mindspore.ops.**arcsinh**(*input*)

Alias for *mindspore.ops.asinh()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.arctan

mindspore.ops.**arctan**(*input*)

Alias for *mindspore.ops.atan()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.arctanh

mindspore.ops.**arctanh**(*input*)

Alias for *mindspore.ops.atanh()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.arctan2

mindspore.ops.**arctan2**(*input*, *other*)

Alias for *mindspore.ops.atan2()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.asin

mindspore.ops.**asin**(*input*)

Computes arcsine of input tensors element-wise.

$$out_i = \sin^{-1}(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.asin(mindspore.tensor([0.74, 0.04, 0.30, 0.56]))
>>> print(output)
[0.8330704  0.04001067  0.30469266  0.5943858 ]
```

mindspore.ops.asinh

mindspore.ops.**asinh**(*input*)

Computes inverse hyperbolic sine of the input element-wise.

$$out_i = \sinh^{-1}(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.asinh(mindspore.tensor([-5.0, 1.5, 3.0, 100.0]))
>>> print(output)
[-2.3124382  1.1947632  1.8184465  5.298342 ]
```

mindspore.ops.atan

mindspore.ops.**atan**(*input*)

Computes the trigonometric inverse tangent of the input element-wise.

$$out_i = \tan^{-1}(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.atan(mindspore.tensor([1.0, 0.0]))
>>> print(output)
[0.7853982 0.       ]
```

mindspore.ops.atan2

mindspore.ops.**atan2**(*input*, *other*)

Returns arctangent of input/other element-wise.

It returns $\theta \in [-\pi, \pi]$ such that $input = r * \sin(\theta), other = r * \cos(\theta)$, where $r = \sqrt{input^2 + other^2}$.

Parameters

- **input** (Tensor, *Number.number*) –The input tensor or scalar.
- **other** (Tensor, *Number.number*) –The input tensor or scalar.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.atan2(mindspore.tensor([0., 1.]), mindspore.tensor([1., 1.]))
>>> print(output)
[0.        0.7853982]
```

mindspore.ops.atanh

mindspore.ops.**atanh**(*input*)

Computes inverse hyperbolic tangent of the input element-wise.

$$out_i = \tanh^{-1}(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.atanh(mindspore.tensor([0, -0.5]))
>>> print(output)
[ 0.         -0.54930615]
```

mindspore.ops.atleast_1d

mindspore.ops.**atleast_1d**(*inputs*)

Returns a one-dimensional tensor of each zero-dimensional tensor, while tensors with one or more dimensions remain unchanged.

Parameters

inputs (*Union[Tensor, list[Tensor]]*) – The input tensor or list of tensors.

Returns

Tensor or list of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: Input is a zero-dimensional tensor.
>>> x = mindspore.tensor(1)
>>> mindspore.ops.atleast_1d(x)
Tensor(shape=[1], dtype=Int64, value= [1])
>>>
>>> # case 2: Input is a one-dimensional tensor.
>>> y = mindspore.tensor([0, 1])
>>> mindspore.ops.atleast_1d(y)
Tensor(shape=[2], dtype=Int64, value= [0, 1])
>>>
>>> # case 3: Input is a list containing tensors of various dimensions.
>>> mindspore.ops.atleast_1d([x, y])
(Tensor(shape=[1], dtype=Int64, value= [1]), Tensor(shape=[2], dtype=Int64, value= [0, 1]))
```

mindspore.ops.atleast_2d

mindspore.ops.**atleast_2d**(*inputs*)

Returns a 2-dimensional tensor of each tensor, while tensors with two or more dimensions remain unchanged.

Parameters

inputs (*Union[Tensor, list[Tensor]]*) – The input tensor or list of tensors.

Returns

Tensor or list of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: Input is a zero-dimensional tensor.
>>> x = mindspore.tensor(1)
>>> mindspore.ops.atleast_2d(x)
Tensor(shape=[1, 1], dtype=Int64, value= [[1]])
>>>
>>> # case 2: Input is a 2-dimensional tensor.
>>> y = mindspore.tensor([[0, 1], [2, 3]])
>>> mindspore.ops.atleast_2d(y)
Tensor(shape=[2, 2], dtype=Int64, value=
[[0, 1],
 [2, 3]])
>>>
>>> # case 3: Input is a list containing tensors of various dimensions.
>>> mindspore.ops.atleast_2d([x, y])
(Tensor(shape=[1, 1], dtype=Int64, value=
 [[1]]),
 Tensor(shape=[2, 2], dtype=Int64, value=
 [[0, 1],
  [2, 3]]))
```

mindspore.ops.atleast_3d

mindspore.ops.**atleast_3d**(*inputs*)

Returns a 3-dimensional tensor of each tensor, while tensors with three or more dimensions remain unchanged.

Parameters

inputs (*Union[Tensor, list[Tensor]]*) – The input tensor or list of tensors.

Returns

Tensor or list of tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: Input is a zero-dimensional tensor.
>>> x = mindspore.tensor(1)
>>> mindspore.ops.atleast_3d(x)
Tensor(shape=[1, 1, 1], dtype=Int64, value= [[[1]]])
>>>
>>> # case 2: Input is a 3-dimensional tensor.
>>> y = mindspore.tensor([[[0, 1], [2, 3]]])
>>> mindspore.ops.atleast_3d(y)
Tensor(shape=[1, 2, 2], dtype=Int64, value=
[[[0, 1],
  [2, 3]]])
>>>
>>> # case 3: Input is a list containing tensors of various dimensions.
>>> mindspore.ops.atleast_3d([x, y])
(Tensor(shape=[1, 1, 1], dtype=Int64, value=
 [[[1]]]),
 Tensor(shape=[1, 2, 2], dtype=Int64, value=
 [[[0, 1],
   [2, 3]]])
```

mindspore.ops.bessel_i0

mindspore.ops.**bessel_i0**(*x*)

Computes the zeroth order modified Bessel function of the first kind for each element input.

$$I_0(x) = J_0(\mathrm{i}x) = \sum_{m=0}^{\infty} \frac{x^{2m}}{2^{2m}(m!)^2}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([-1., -0.5, 0.5, 1.])
>>> output = mindspore.ops.bessel_i0(x)
>>> print(output)
[1.266066  1.0634835 1.0634835 1.266066]
```

mindspore.ops.bessel_i0e

mindspore.ops.**bessel_i0e**(*x*)

Computes the exponentially scaled zeroth order modified Bessel function of the first kind for each element input.

$$\mathrm{I}_0e(x) = e^{(-|x|)} * I_0(x) = e^{(-|x|)} * \sum_{m=0}^{\infty} \frac{x^{2m}}{2^{2m}(m!)^2}$$

Parameters

x (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([-1., -0.5, 0.5, 1.])
>>> output = mindspore.ops.bessel_i0e(x)
>>> print(output)
[0.46575961  0.64503527  0.64503527  0.46575961]
```

mindspore.ops.bessel_i1

mindspore.ops.**bessel_i1**(*x*)

Computes the first order modified Bessel function of the first kind for each element input.

$$I_1(x) = \mathrm{i}^{-1}J_1(\mathrm{i}x) = \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2^{2m+1}m!(m+1)!}$$

Parameters

x (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([-1., -0.5, 0.5, 1.])
>>> output = mindspore.ops.bessel_i1(x)
>>> print(output)
[-0.5651591  -0.25789431  0.25789431  0.5651591]
```

mindspore.ops.bessel_i1e

mindspore.ops.**bessel_i1e**(*x*)

Computes the exponentially scaled first order modified Bessel function of the first kind for each element input.

$$\mathrm{I}_1e(x) = e^{(-|x|)} * I_1(x) = e^{(-|x|)} * \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2^{2m+1}m!(m+1)!}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([-1., -0.5, 0.5, 1.])
>>> output = mindspore.ops.bessel_i1e(x)
>>> print(output)
[-0.20791042  -0.15642083  0.15642083  0.20791042]
```

mindspore.ops.bessel_j0

mindspore.ops.**bessel_j0**(*x*)

Computes the zeroth order Bessel function of the first kind for each element input.

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi}\cos(x\sin\theta)d\theta = \sum_{m=0}^{\infty}\frac{(-1)^m x^{2m}}{2^{2m}(m!)^2}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_j0(x)
>>> print(output)
[0.93846981  0.76519769  0.22389078  -0.39714981]
```

mindspore.ops.bessel_j1

mindspore.ops.**bessel_j1**(*x*)

Computes the first order Bessel function of the first kind for each element input.

$$J_1(x) = \frac{1}{\pi}\int_0^{\pi}\cos(x\sin\theta-\theta)d\theta = \sum_{m=0}^{\infty}\frac{(-1)^m x^{2m+1}}{2^{2m+1}m!(m+1)!}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_j1(x)
>>> print(output)
[0.24226846  0.44005059  0.57672481 -0.06604333]
```

mindspore.ops.bessel_k0

mindspore.ops.**bessel_k0**(*x*)

Computes the zeroth order modified Bessel function of the second kind for each element input.

$$K_0(x) = \lim_{\nu\to 0}\left(\frac{\pi}{2}\right)\frac{I_{-\nu}(x)-I_{\nu}(x)}{\sin(\nu\pi)} = \int_0^{\infty} e^{-x\cosh t}dt$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_k0(x)
>>> print(output)
[0.92441907  0.42102444  0.11389387  0.01115968]
```

mindspore.ops.bessel_k0e

mindspore.ops.**bessel_k0e**(*x*)

Computes the exponentially scaled zeroth order modified Bessel function of the second kind for each element input.

$$K_0e(x) = e^{(-|x|)} * K_0(x) = e^{(-|x|)} * \int_0^\infty e^{-x\cosh t} dt$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_k0e(x)
>>> print(output)
[1.52410939  1.14446308  0.84156822  0.60929767]
```

mindspore.ops.bessel_k1

mindspore.ops.**bessel_k1**(*x*)

Computes the first order modified Bessel function of the second kind for each element input.

$$K_1(x) = \lim_{\nu\to 1}\left(\frac{\pi}{2}\right)\frac{I_{-\nu}(x)-I_\nu(x)}{\sin(\nu\pi)} = \int_0^\infty e^{-x\cosh t}\cosh(t)dt$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_k1(x)
>>> print(output)
[1.65644112  0.60190723  0.13986588  0.0124835]
```

mindspore.ops.bessel_k1e

mindspore.ops.**bessel_k1e**(*x*)

Computes the exponentially scaled first order modified Bessel function of the second kind for each element input.

$$K_1e(x) = e^{(-|x|)} * K_1(x) = e^{(-|x|)} * \int_0^\infty e^{-x\cosh t}\cosh(t)dt$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_k1e(x)
>>> print(output)
[2.73100971  1.63615349  1.03347685  0.68157595]
```

mindspore.ops.bessel_y0

mindspore.ops.**bessel_y0**(*x*)

Computes the zeroth order Bessel function of the second kind for each element input.

$$Y_0(x) = \lim_{n\to 0} \frac{J_n(x)\cos n\pi - J_{-n}(x)}{\sin n\pi}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_y0(x)
>>> print(output)
[-0.44451874  0.08825696  0.51037567  -0.01694074]
```

mindspore.ops.bessel_y1

mindspore.ops.**bessel_y1**(*x*)

Computes the first order Bessel function of the second kind for each element input.

$$Y_1(x) = \lim_{n \to 1} \frac{J_n(x)\cos n\pi - J_{-n}(x)}{\sin n\pi}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.5, 1., 2., 4.])
>>> output = mindspore.ops.bessel_y1(x)
>>> print(output)
[-1.47147239  -0.78121282  -0.10703243  0.39792571]
```

mindspore.ops.bitwise_and

mindspore.ops.**bitwise_and**(*input*, *other*)

Compute the bitwise AND of two input tensors. If *input* and *other* have different data types, the implicit type conversion rules and type promotion rules are followed.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, -2, 3])
>>> other = mindspore.tensor([1, 0, 3])
>>> mindspore.ops.bitwise_and(input, other)
Tensor(shape=[3], dtype=Int64, value= [1, 0, 3])
>>> # Same as calling via the | operator:
>>> input & other
Tensor(shape=[3], dtype=Int64, value= [1, 0, 3])
```

mindspore.ops.bitwise_left_shift

mindspore.ops.**bitwise_left_shift**(*input*, *other*)

Perform a left bitwise shift operation on the *input* element-wise, where the number of bits to shift is specified by *other*.

$$out_i = input_i << other_i$$

Parameters

- **input** (*Union[Tensor, int, bool]*) – The input to be left shifted.
- **other** (*Union[Tensor, int, bool]*) – The number of bit to be applied on left arithmetic shift.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 4, 8])
>>> mindspore.ops.bitwise_left_shift(input, 1)
Tensor(shape=[4], dtype=Int64, value= [ 2,  4,  8, 16])
```

mindspore.ops.bitwise_or

mindspore.ops.**bitwise_or**(*input*, *other*)

Compute the bitwise OR of two input tensors. If *input* and *other* have different data types, the implicit type conversion rules and type promotion rules are followed.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, -2, 3])
>>> other = mindspore.tensor([1, 0, 3])
>>> mindspore.ops.bitwise_or(input, other)
Tensor(shape=[3], dtype=Int64, value= [-1, -2,  3])
>>> # Same as calling via the | operator:
>>> input | other
Tensor(shape=[3], dtype=Int64, value= [-1, -2,  3])
```

mindspore.ops.bitwise_right_shift

mindspore.ops.**bitwise_right_shift**(*input*, *other*)

Perform a right bitwise shift operation on the *input* element-wise, where the number of bits to shift is specified by *other*.

$$out_i = input_i >> other_i$$

Parameters

- **input** (*Union[Tensor, int, bool]*) – The input to be right shifted.
- **other** (*Union[Tensor, int, bool]*) – The number of bit to be applied on right arithmetic shift.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 4, 8])
>>> mindspore.ops.bitwise_right_shift(input, 1)
Tensor(shape=[4], dtype=Int64, value= [0, 1, 2, 4])
```

mindspore.ops.bitwise_xor

mindspore.ops.**bitwise_xor**(*input*, *other*)

Compute the bitwise XOR of two input tensors. If *input* and *other* have different data types, the implicit type conversion rules and type promotion rules are followed.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, -2, 3])
>>> other = mindspore.tensor([1, 0, 3])
>>> mindspore.ops.bitwise_xor(input, other)
Tensor(shape=[3], dtype=Int64, value= [-2, -2,  0])
>>> # Same as calling via the ^ operator:
>>> input ^ other
Tensor(shape=[3], dtype=Int64, value= [-2, -2,  0])
```

mindspore.ops.ceil

mindspore.ops.**ceil**(*input*)

Rounds a tensor up to the closest integer element-wise.

$$out_i = \lceil input_i \rceil$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(([1.1, 2.5, -1.5]))
>>> output = mindspore.ops.ceil(input)
>>> print(output)
[ 2.  3. -1.]
```

mindspore.ops.clamp

mindspore.ops.**clamp**(*input*, *min=None*, *max=None*)

Clamp all elements of the input tensor within the range [min, max].

$$out_i = \begin{cases} max & \text{if } input_i \geq max \\ input_i & \text{if } min < input_i < max \\ min & \text{if } input_i \leq min \end{cases}$$

Note:

- *min* and *max* cannot be None at the same time;
- If *min* is None , there is no lower bound.
- if *max* is None , there is no upper bound.

- If *min* is greater than *max*, the value of all elements in Tensor will be set to *max*;

Parameters

- **input** (Tensor) –The input tensor.
- **min** (*Union(Tensor, float, int), optional*) –The minimum value. Default None .
- **max** (*Union(Tensor, float, int), optional*) –The maximum value. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: `min` and `max` are integer
>>> input = mindspore.tensor([[1, 25, 5, 7], [4, 11, 6, 21]])
>>> mindspore.ops.clamp(input, 5, 20)
Tensor(shape=[2, 4], dtype=Int64, value=
[[ 5, 20,  5,  7],
 [ 5, 11,  6, 20]])
>>>
>>> # case 2: If `min` and `max` are tensors, their shapes need to be broadcastable with␣
↪input.
>>> min = mindspore.tensor([2, 4, 6, 8])
>>> max = mindspore.tensor([10, 12, 14, 18])
>>> mindspore.ops.clamp(input, min, max)
Tensor(shape=[2, 4], dtype=Int64, value=
[[ 2, 12,  6,  8],
 [ 4, 11,  6, 18]])
```

mindspore.ops.clip

mindspore.ops.**clip**(*input*, *min=None*, *max=None*)

Alias for *mindspore.ops.clamp()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.combinations

mindspore.ops.**combinations**(*input*, *r=2*, *with_replacement=False*)

Return all r-length subsequences of input tensor.

When *with_replacement* is set to `False`, it works similar to Python's *itertools.combinations*, and when *with_replacement* is set to `True`, it behaves like *itertools.combinations_with_replacement*.

Parameters

- **input** (Tensor) –One-dimensional input tensor.
- **r** (*int, optional*) –Number of elements to perform combination. Default `2` .
- **with_replacement** (*bool, optional*) –Allow duplication or not. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3])
>>> mindspore.ops.combinations(input)
Tensor(shape=[3, 2], dtype=Int64, value=
[[1, 2],
 [1, 3],
 [2, 3]])
>>> mindspore.ops.combinations(input, r=3)
Tensor(shape=[1, 3], dtype=Int64, value=
[[1, 2, 3]])
>>> mindspore.ops.combinations(input, with_replacement=True)
Tensor(shape=[6, 2], dtype=Int64, value=
[[1, 1],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [3, 3]])
>>> # It has the same results as using itertools.combinations.
>>> import itertools
>>> input = [1, 2, 3]
>>> list(itertools.combinations(input, r=2))
[(1, 2), (1, 3), (2, 3)]
>>> list(itertools.combinations(input, r=3))
[(1, 2, 3)]
>>> list(itertools.combinations_with_replacement(input, r=2))
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

mindspore.ops.copysign

mindspore.ops.**copysign**(*x*, *other*)

Create a float tensor composed of the absolute values of *x* and the signs of *other* . Support broadcasting.

Parameters

- **x** (*Union[Tensor]*) –The input tensor.
- **other** (*Union[int, float, Tensor]*) –A tensor that determines the sign of the return value.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: When other is a tensor
>>> x = mindspore.tensor([[0.3, -0.7],
...                       [0.5, 0.5]])
>>> other = mindspore.tensor([-0.4, 0.6])
>>> output = mindspore.ops.copysign(x, other)
>>> print(output)
[[-0.3  0.7]
 [-0.5  0.5]]
>>> # case 2: When other is a scalar
>>> output = mindspore.ops.copysign(x, 2)
>>> print(output)
[[0.3 0.7]
 [0.5 0.5]]
```

mindspore.ops.cos

mindspore.ops.**cos**(*input*)

Computes cosine of input element-wise.

$$out_i = \cos(x_i)$$

Warning: Using float64 may cause a problem of missing precision.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [bool, int8, uint8,
 ↪ int16, int32, int64](Datatype only supported on Ascend).
>>> input = mindspore.tensor([0, 1, 2], mindspore.int32)
>>> mindspore.ops.cos(input)
Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  5.40302306e-01, -4.16146837e-01])
>>>
>>> # Otherwise output has the same dtype as the `input`.
>>> input = mindspore.tensor([0.74, 0.04, 0.30, 0.56], mindspore.float64)
>>> mindspore.ops.cos(input)
Tensor(shape=[4], dtype=Float64, value= [ 7.38468559e-01,  9.99200107e-01,  9.55336489e-01,
 ↪8.47255111e-01])
```

mindspore.ops.cosh

mindspore.ops.**cosh**(*input*)

Computes hyperbolic cosine of input element-wise.

$$out_i = \cosh(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.74, 0.04, 0.30, 0.56])
>>> output = mindspore.ops.cosh(input)
>>> print(output)
[1.2865248 1.0008001 1.0453385 1.1609408]
```

mindspore.ops.cosine_similarity

mindspore.ops.**cosine_similarity**(*x1*, *x2*, *dim=1*, *eps=1e-08*)

Calculate cosine similarity between two input tensors along the specified dimension.

$$\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}$$

Note: Currently, broadcast of input is not supported.

Parameters

- **x1** (Tensor) –The first input tensor.
- **x2** (Tensor) –The second input tensor.
- **dim** (*int, optional*) –Specify the dimension for computation. Default 1 .
- **eps** (*float, optional*) –Minimal value to avoid division by zero. Default 1e-08 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-0.0256, 0.0127, -0.2475, 0.2316, 0.8037],
...                           [0.5809, -1.2712, -0.7038, -0.2558, 0.7494]])
>>> other = mindspore.tensor([[-0.6115, -0.1965, -0.8484, 0.2389, 0.2409],
...                           [1.8940, -2.1997, 0.1915, 0.0856, 0.7542]])
>>> output = mindspore.ops.cosine_similarity(input, other)
>>> print(output)
[0.4843164  0.81647635]
```

mindspore.ops.cov

mindspore.ops.**cov**(*input*, *, *correction=1*, *fweights=None*, *aweights=None*)

Return the covariance matrix of the input tensor, where the rows of the input tensor represent variables and the columns represent observations. The diagonal of the covariance matrix contains the variance of each variable in the input tensor, while the off-diagonal elements represent the covariance between pairs of variables. If the input is a 0-D or 1-D tensor, its variance is returned.

The unbiased sample covariance of the variables a and b is given by the following formula:

$$\text{cov}_w(a, b) = \frac{\sum_{i=1}^{N}(a_i - \bar{a})(b_i - \bar{b})}{N - 1}$$

where $\bar{a}$ and $\bar{b}$ are the simple means of the a and b respectively.

If *fweights* and/or *aweights* are provided, the unbiased weighted covariance is calculated, which is given by:

$$\text{cov}_w(a, b) = \frac{\sum_{i=1}^{N} w_i(a_i - \mu_a^*)(b_i - \mu_b^*)}{\sum_{i=1}^{N} w_i - 1}$$

where w denotes *fweights* or *aweights* based on whichever is provided, or $w = fweights \times aweights$ if both are provided, and $\mu_x^* = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i}$ is the weighted mean of the variable.

Warning: The values of *fweights* and *aweights* cannot be negative, and the negative weight scene result is undefined.

Note: Currently, complex number is not supported.

Parameters

input (Tensor) –The 0-D, 1-D or 2-D tensor.

Keyword Arguments

- **correction** (int, *optional*) –The difference between sample size and sample degrees of freedom. *correction = 0* will return the simple average. Defaults to Bessel's correction, *correction = 1* which returns the unbiased estimate, even if both *fweights* and *aweights* are specified.
- **fweights** (Tensor, *optional*) –0D or 1D tensor containing integer frequency weight, indicating the number of repetition of each observation vector. Its numel must equal the number of columns of *input*. Ignored if *None*. Default None .
- **aweights** (Tensor, *optional*) –A scalar or 1D Tensor containing float observation weights represents the importance of each observation vector. The higher the importance, the greater the corresponding value. Its numel must equal the number of columns of *input*. Must have floating point dtype. Ignored if *None*. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[0., 5., 7.],
...                           [3., 5., 0.]])
>>> output = mindspore.ops.cov(input)
>>> print(output)
[[13.         -3.5       ]
 [-3.5         6.3333335]]
>>> output = mindspore.ops.cov(input, correction=0)
>>> print(output)
[[ 8.666667  -2.3333333]
 [-2.3333333  4.2222223]]
>>> fw = mindspore.tensor([5, 2, 4], dtype=mindspore.int64)
>>> aw = mindspore.tensor([0.4588, 0.9083, 0.7616], mindspore.float32)
>>> output = mindspore.ops.cov(input, fweights=fw, aweights=aw)
>>> print(output)
[[10.146146 -3.47241 ]
 [-3.47241   4.716825]]
```

mindspore.ops.diff

mindspore.ops.**diff**(*x*, *n=1*, *axis=- 1*, *prepend=None*, *append=None*)

Computes the n-th forward difference along the given axis.

The first-order differences is calculated as $out[i] = x[i+1] - x[i]$. Higher-order differences are calculated by using *mindspore.ops.diff()* recursively.

Note: Zero-shaped Tensor is not supported, a value error is raised if an empty Tensor is encountered. Any dimension of a Tensor is 0, which is considered an empty Tensor. Tensor with shape of $(0,)$, $(1, 2, 0, 4)$ are all empty Tensor.

Parameters

- **x** (Tensor) −The input tensor.
- **n** (*int, optional*) −The number of times to compute the difference. Currently only 1 is supported. Default 1 .
- **axis** (*int, optional*) −The axis to compute the difference along. Default −1 .
- **prepend** (Tensor, *optional*) −Values to prepend to *x* along *axis* before performing the difference. Their dimensions must be equivalent to that of *x*, and their shapes must match input's shape except on dim. Default None .
- **append** (Tensor, *optional*) −Values to append to *x* along *axis* before performing the difference. Their dimensions must be equivalent to that of *x*, and their shapes must match input's shape except on dim. Default None .

Returns

Tensor, the n-th differences of input. The shape of the output is the same as *x* except along *axis* where the size is reduced by *n*. The type of the output is the same as *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 3, 2])
>>> # case 1: By default, compute the first-order differences along axis -1.
>>> mindspore.ops.diff(x)
Tensor(shape=[2], dtype=Int64, value= [ 2, -1])
>>>
>>> # case 2: When argument prepend is setting:
>>> n = mindspore.tensor([4, 5])
>>> mindspore.ops.diff(x, prepend=n)
Tensor(shape=[4], dtype=Int64, value= [ 1, -4,  2, -1])
>>>
>>> # case 3: When argument append is setting:
>>> mindspore.ops.diff(x, append=n)
Tensor(shape=[4], dtype=Int64, value= [ 2, -1,  2,  1])
>>>
>>> # case 4: When input is 2-D dimensional tensor, compute forward difference along␣
 ↪different axis.
>>> x = mindspore.tensor([[1, 2, 3], [3, 4, 5]])
>>> mindspore.ops.diff(x, axis=0)
Tensor(shape=[1, 3], dtype=Int64, value=
[[2, 2, 2]])
>>> mindspore.ops.diff(x, axis=1)
Tensor(shape=[2, 2], dtype=Int64, value=
[[1, 1],
 [1, 1]])
```

mindspore.ops.deg2rad

mindspore.ops.**deg2rad**(*x*)

Convert angles from degrees to radians element-wise.

Parameters

x (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[90.0, -90.0], [180.0, -180.0], [270.0, -270.0]])
>>> output = mindspore.ops.deg2rad(input)
>>> print(output)
[[ 1.5707964 -1.5707964]
 [ 3.1415927 -3.1415927]
 [ 4.712389  -4.712389 ]]
```

mindspore.ops.digamma

mindspore.ops.**digamma**(*input*)

Computes the logarithmic derivative of the gamma function on input tensor.

$$P(x) = \frac{d}{dx}(\ln(\Gamma(x)))$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.5, 0.5, 9])
>>> output = mindspore.ops.digamma(input)
>>> print(output)
[ 0.03648992 -1.9635109   2.1406415 ]
```

mindspore.ops.div

mindspore.ops.**div**(*input*, *other*, *, *rounding_mode=None*)

Divide the first input tensor by the second input tensor in floating-point type element-wise.

$$out_i = input_i / other_i$$

Note:

- The two input tensors must be broadcastable.
- The two input tensors can not be bool type at the same time,
- The two input tensors comply with the implicit type conversion rules to make the data types consistent.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input tensor.
- **other** (*Union[Tensor, Number, bool]*) – The second input tensor.

Keyword Arguments

rounding_mode (*str, optional*) – Type of rounding applied to the result. Default `None` . Three types are defined as:

- None: the same as true division in Python or *true_divide* in NumPy.
- "floor": rounds the results of the division down, which is the same as floor division in Python or *floor_divide* in NumPy.
- "trunc": rounds the results of the division towards zero, which is the same as C-style integer division.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-0.3711, -1.9353, -0.4605, -0.2917],
...                           [ 0.1815, -1.0111,  0.9805, -1.5923],
...                           [ 0.1062,  1.4581,  0.7759, -1.2344],
...                           [-0.1830, -0.0313,  1.1908, -1.4757]])
>>> other = mindspore.tensor([ 0.8032,  0.2930, -0.8113, -0.2308])
>>> output = mindspore.ops.div(input, other)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[-0.4620269  -6.605119    0.5676076   1.2638649 ]
 [ 0.22597112 -3.4508533  -1.2085541   6.899047  ]
 [ 0.13222112  4.97645    -0.95636636  5.348354  ]
 [-0.22783864 -0.10682594 -1.4677677   6.3938475 ]]
>>> output = mindspore.ops.div(input, other, rounding_mode='floor')
>>> print(output)
[[-1. -7.  0.  1.]
 [ 0. -4. -2.  6.]
 [ 0.  4. -1.  5.]
 [-1. -1. -2.  6.]]
```

mindspore.ops.divide

mindspore.ops.**divide**(*input*, *other*, *, *rounding_mode=None*)

Alias for *mindspore.ops.div()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.erf

mindspore.ops.**erf**(*input*)

Compute the Gauss error of input tensor element-wise.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [int64,␣
↪bool](Datatype only supported on Ascend).
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.int64)
>>> mindspore.ops.erf(input)
Tensor(shape=[5], dtype=Float32, value= [-8.42700793e-01,  0.00000000e+00,  8.42700793e-01, ␣
↪9.95322265e-01,  9.99977910e-01])
>>>
>>> # Otherwise output has the same dtype as the input.
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.float64)
```

(continues on next page)

(continued from previous page)

```
>>> mindspore.ops.erf(input)
Tensor(shape=[5], dtype=Float64, value= [-8.42700793e-01,  0.00000000e+00,  8.42700793e-01, ␣
↪9.95322265e-01,  9.99977910e-01])
```

mindspore.ops.erfc

mindspore.ops.**erfc**(*input*)

Compute the complementary error function of input tensor element-wise.

$$\operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [int64,␣
↪bool](Datatype only supported on Ascend).
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.int64)
>>> mindspore.ops.erfc(input)
Tensor(shape=[5], dtype=Float32, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
>>>
>>> # Otherwise output has the same dtype as the input.
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.float64)
>>> mindspore.ops.erfc(input)
Tensor(shape=[5], dtype=Float64, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
```

mindspore.ops.erfinv

mindspore.ops.**erfinv**(*input*)

Compute the inverse error of input tensor element-wise.

It is defined in the range *(-1, 1)* as:

$$erfinv(erf(x)) = x$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # When the `input` is int8, int16, int32, int64, uint8 or bool, the return value type is␣
↪float32.
>>> input = mindspore.tensor([0, 0.5, -0.9], mindspore.int64)
>>> mindspore.ops.erfinv(input)
Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00])
>>> # Otherwise, the return value type is the same as the input type.
>>> input = mindspore.tensor([0, 0.5, -0.9], mindspore.float32)
>>> mindspore.ops.erfinv(input)
Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  4.76936132e-01, -1.16308689e+00])
```

mindspore.ops.exp

mindspore.ops.**exp**(*input*)

Compute exponential of the input tensor element-wise.

$$out_i = e^{x_i}$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.exp(input)
>>> print(output)
[ 1.        2.7182817 20.085537]
```

mindspore.ops.exp2

mindspore.ops.**exp2**(*input*)

Compute base two exponential of the input tensor element-wise.

$$out_i = 2^{input_i}$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.exp2(input)
>>> print(output)
[1. 2. 8.]
```

mindspore.ops.expm1

mindspore.ops.**expm1**(*input*)

Compute exponential of the input tensor, then minus 1, element-wise.

$$out_i = e^{x_i} - 1$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.expm1(input)
>>> print(output)
[ 0.         1.7182817 19.085537 ]
```

mindspore.ops.floor

mindspore.ops.**floor**(*input*)

Rounds a tensor down to the closest integer element-wise.

$$out_i = \lfloor input_i \rfloor$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.1, 2.5, -1.5])
>>> output = mindspore.ops.floor(input)
>>> print(output)
[ 1.  2. -2.]
```

mindspore.ops.floor_div

mindspore.ops.**floor_div**(*x*, *y*)

Alias for *mindspore.ops.floor_divide()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.floor_divide

mindspore.ops.**floor_divide**(*input*, *other*)

Compute element-wise division of *input* by *other* and floor the result.

If *input* and *other* have different data types, the implicit type conversion rules are followed. Inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, their shapes must be broadcastable, and their data types cannot both be bool simultaneously.

$$out_i = \text{floor}(\frac{input_i}{other_i})$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input tensor.

- **other** (*Union[Tensor, Number, bool]*) – The second input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: Two tensors with boolean and integer data type.
>>> input = mindspore.tensor([True, True, False])
>>> other = mindspore.tensor([1, 2, 4])
>>> output = mindspore.ops.floor_divide(input, other)
>>> print(output)
[1 0 0]
>>>
>>> # case 2: One tensor and one scalar.
>>> input = mindspore.tensor([1, 2, 4])
>>> other = mindspore.tensor(1.5)
>>> output = mindspore.ops.floor_divide(input, other)
>>> print(output)
[0. 1. 2.]
>>>
>>> # case 3: When inputs have different data types, type promotion rules are followed.
>>> input = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> other = mindspore.tensor([1.1, 2.5, -1.5], mindspore.float32)
>>> output = mindspore.ops.floor_divide(input, other)
>>> print(output)
[ 0.  0. -3.]
```

mindspore.ops.floor_mod

`mindspore.ops.`**`floor_mod`**(*x*, *y*)

Compute the remainder of element-wise flooring division of first input by second input.

If two input have different data types, implicit type conversion rules are followed. Inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, their shapes must be broadcastable, and their data types cannot both be bool simultaneously.

$$out_i = \text{floor}(x_i // y_i)$$

Warning:

- Data of input *y* should not be 0, or the maximum value of its dtype will be returned.
- When the elements of input exceed 2048, the accuracy of operator cannot guarantee the requirement of double thousandths in the mini form.
- Due to different architectures, the calculation results of this operator on NPU and CPU may be inconsistent.
- If shape is expressed as $(D1, D2..., Dn)$, then D1*D2$\cdots$*DN<=1000000,n<=8.

Parameters

- **x** (*Union[Tensor, Number, bool]*) – The first input tensor.
- **y** (*Union[Tensor, Number, bool]*) – The second input tensor.

Returns

Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: Two tensors with boolean and integer data type.
>>> input = mindspore.tensor([True, True, False])
>>> other = mindspore.tensor([1, 2, 4])
>>> output = mindspore.ops.floor_mod(input, other)
>>> print(output)
[0 1 0]
>>>
>>> # case 2: One tensor and one scalar.
>>> input = mindspore.tensor([1, 2, 4])
>>> other = mindspore.tensor(1.5)
>>> output = mindspore.ops.floor_mod(input, other)
>>> print(output)
[1.  0.5 1. ]
>>>
>>> # case 3: When inputs have different data types, type promotion rules are followed.
>>> input = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> other = mindspore.tensor([1.1, 2.5, -1.5], mindspore.float32)
>>> output = mindspore.ops.floor_mod(input, other)
>>> print(output)
[ 1.   2.  -0.5]
```

mindspore.ops.float_power

mindspore.ops.**float_power**(*input*, *exponent*)

Computes the first input to the power of the second input. For the real number type, cast *input* and *exponent* to mindspore.float64 to calculate.

Note: Currently, complex type calculation is not supported.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **exponent** (*Union[Tensor, Number]*) – The second input, if the first input is Number, it must be a Tensor.

Returns

Tensor whose data type is mindspore.float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: When exponent is scalar:
>>> input = mindspore.tensor([-1.5, 0., 2.])
>>> output = mindspore.ops.float_power(input, 2)
>>> print(output)
[2.25 0.   4.  ]
>>>
>>> # case 2: When exponent is a tensor:
>>> exponent = mindspore.tensor([0, 1, 2])
>>> output = mindspore.ops.float_power(input, exponent)
>>> print(output)
[1. 0. 4.]
```

mindspore.ops.fmax

mindspore.ops.**fmax**(*input*, *other*)

Compute the maximum of input tensors element-wise.

$$output_i = \max(x1_i, x2_i)$$

Note:

- Support implicit type conversion and type promotion.
- Shapes of *input* and *other* should be able to broadcast.
- If one of the elements to be compared is NaN, another element is returned.

Parameters

- **input** (Tensor) –The first input.
- **other** (Tensor) –The second input.

Returns

Tensor

Supported Platforms:

CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1.0, 5.0, 3.0], mindspore.float32)
>>> x2 = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float32)
>>> output = mindspore.ops.fmax(x1, x2)
>>> print(output)
[4. 5. 6.]
```

mindspore.ops.fmod

mindspore.ops.**fmod**(*input*, *other*)

Compute the remainder of element-wise division of first input by the second input.

Support broadcasting and type promotion.

$$out = input - n * other$$

Where n is $input/other$ with its fractional part truncated. The returned value has the same sign as *input* and is less than *other* in magnitude.

Parameters

- **input** (*Union[Tensor, Number]*) – the dividend.
- **other** (*Union[Tensor, Number]*) – the divisor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: When other is scalar:
>>> input = mindspore.tensor([-3., -2, -1, 3, 4, 5])
>>> output = mindspore.ops.fmod(input, 2)
>>> print(output)
[-1.  0. -1.  1.  0.  1.]
>>>
>>> # case 2: When other is a tensor:
>>> other = mindspore.tensor([-1., 1, 2, 2.5, 1.5, 1.7])
>>> output = mindspore.ops.fmod(input, other)
>>> print(output)
[ 0.         0.        -1.         0.5        1.         1.5999999]
```

mindspore.ops.frac

mindspore.ops.**frac**(*x*)

Return the fractional part of each element in the input tensor.

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([2, 4.2, -2.5])
>>> output = mindspore.ops.frac(input)
>>> print(output)
[ 0.          0.19999981 -0.5       ]
```

mindspore.ops.gcd

mindspore.ops.**gcd**(*input*, *other*)

Computes greatest common divisor of input tensors element-wise.

Support broadcasting and type promotion. Data types should be one of: int16 (supported when using the Ascend backend, Graph mode is only supported when the graph compilation level is O0), int32, int64.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([7, 8, 9])
>>> other = mindspore.tensor([14, 6, 12])
>>> mindspore.ops.gcd(input, other)
Tensor(shape=[3], dtype=Int64, value= [7, 2, 3])
```

mindspore.ops.hypot

mindspore.ops.**hypot**(*input*, *other*)

Given the legs of a right triangle, return its hypotenuse, element-wise.

Support broadcasting and type promotion.

$$out_i = \sqrt{input_i^2 + other_i^2}$$

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([3., 5., 7.])
>>> other = mindspore.tensor([4., 12., 24.])
>>> output = mindspore.ops.hypot(input, other)
>>> print(output)
[ 5. 13. 25.]
```

mindspore.ops.igamma

mindspore.ops.**igamma**(*input*, *other*)

Calculates lower regularized incomplete Gamma function.

If we define *input* as a and *other* as x, the lower regularized incomplete Gamma function is defined as:

$$P(a,x) = Gamma(a,x)/Gamma(a) = 1 - Q(a,x)$$

where

$$Gamma(a,x) = \int_0^x t^{a-1} \exp^{-t}\, dt$$

is the lower incomplete Gamma function.

Above $Q(a, x)$ is the upper regularized complete Gamma function.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([2.0, 4.0, 6.0, 8.0])
>>> other = mindspore.tensor([2.0, 3.0, 4.0, 5.0])
>>> output = mindspore.ops.igamma(input, other)
>>> print(output)
[0.5939941  0.35276785 0.21486954 0.13337176]
```

mindspore.ops.igammac

mindspore.ops.**igammac**(*input*, *other*)

Calculates upper regularized incomplete Gamma function.

If we define *input* as a and *other* as x, the upper regularized incomplete Gamma function is defined as:

$$Q(a, x) = Gamma(a, x)/Gamma(a) = 1 - P(a, x)$$

where

$$Gamma(a, x) = \int_x^\infty t^{a-1} exp(-t) dt$$

is the upper incomplete Gama function.

Above $P(a, x)$ is the lower regularized incomplete Gamma function.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([2.0, 4.0, 6.0, 8.0])
>>> other = mindspore.tensor([2.0, 3.0, 4.0, 5.0])
>>> output = mindspore.ops.igammac(input, other)
>>> print(output)
[0.40600574 0.6472322  0.78513044 0.8666282 ]
```

mindspore.ops.imag

mindspore.ops.**imag**(*input*)

Return a new tensor containing imaginary value of the input tensor, element-wise. If element in the input tensor is real, it will return zero.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.22+0.3511j, -0.55-0.6796j, -1.92-0.033j, -0.38-0.2991j])
>>> output = mindspore.ops.imag(input)
>>> print(output)
[ 0.3511 -0.6796 -0.033  -0.2991]
```

mindspore.ops.i0

mindspore.ops.**i0**(*input*)

For details, please refer to *mindspore.ops.bessel_i0()*. The parameter *input* of the current interface is the same as the parameter *x* of the reference interface.

Supported Platforms:

GPU CPU

mindspore.ops.inv

mindspore.ops.**inv**(*x*)

Computes Reciprocal of input tensor element-wise.

$$out_i = \frac{1}{x_i}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.25, 0.4, 0.31, 0.52])
>>> output = mindspore.ops.inv(input)
>>> print(output)
[4.        2.5       3.2258065 1.923077 ]
```

mindspore.ops.invert

mindspore.ops.**invert**(*x*)

Flip all bits of input tensor element-wise. For example, 01010101 becomes 10101010.

$$out_i = \sim x_i$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.invert(mindspore.tensor([25, 4, 13, 9], mindspore.int16))
Tensor(shape=[4], dtype=Int16, value= [-26,  -5, -14, -10])
```

mindspore.ops.lcm

mindspore.ops.**lcm**(*input*, *other*)

Computes least common multiplier of input tensors element-wise.

Support broadcast, support implicit type conversion and type promotion.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.lcm(mindspore.tensor([7, 8, 9]), mindspore.tensor([14, 6, 12]))
Tensor(shape=[3], dtype=Int64, value= [14, 24, 36])
```

mindspore.ops.ldexp

mindspore.ops.**ldexp**(*x*, *other*)

Multiplies input tensor by 2^{other} element-wise.

Typically this function is used to construct floating point numbers by multiplying mantissas in *x* with integral powers of two created from the exponents in *other*:

$$out_i = x_i * (2^{other_i})$$

Parameters

- **x** (Tensor) –Mantissas tensor.
- **other** (Tensor) –Exponents tensor, typically integers.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.], mindspore.float32)
>>> other = mindspore.tensor([1, 2, 3, 4], mindspore.int32)
>>> out = mindspore.ops.ldexp(x, other)
>>> print(out)
[ 2.  4.  8. 16.]
>>> x = mindspore.tensor([[1.], [2]], mindspore.float32)
>>> other = mindspore.tensor([[1.], [2]], mindspore.int32)
>>> out = mindspore.ops.ldexp(x, other)
>>> print(out)
[[2.]
 [8.]]
```

mindspore.ops.lerp

mindspore.ops.**lerp**(*input*, *end*, *weight*)

Does a linear interpolation of two tensors input and end based on a float or tensor weight.

$$output_i = input_i + weight_i * (end_i - input_i)$$

Note:

- The shapes of *input* and *end* must be broadcastable.
- If weight is a tensor, then the shapes of *weight* , *start* , and *end* must be broadcastable.
- On the Ascend platform, if *weight* dtype is float, the type of *input* and *end* need to be float32.

Parameters

- **input** (Tensor) –The tensor with the starting points.
- **end** (Tensor) –The tensor with the ending points.
- **weight** (*Union[float, Tensor]*) –The weight for the interpolation formula.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> start = mindspore.tensor([1., 2., 3., 4.], mindspore.float32)
>>> end = mindspore.tensor([10., 10., 10., 10.], mindspore.float32)
>>> output = mindspore.ops.lerp(start, end, 0.5)
>>> print(output)
[5.5 6.  6.5 7. ]
>>> output = mindspore.ops.lerp(start, end, mindspore.tensor([0.5, 0.5, 0.5, 0.5], mindspore.
↪float32))
>>> print(output)
[5.5 6.  6.5 7. ]
```

mindspore.ops.log

mindspore.ops.**log**(*input*)

Compute the natural logarithm of the input tensor element-wise.

$$y_i = \log_e(x_i)$$

Warning: If the input value of operator Log is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.ops.log(x)
>>> print(output)
[0.        0.6931472 1.3862944]
```

mindspore.ops.log2

mindspore.ops.**log2**(*input*)

Compute the logarithm to the base 2 of the input tensor element-wise.

$$y_i = \log_2(input_i)$$

Warning: If the input value of operator log2 is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affected.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([2, 4, 8], mindspore.float16)
>>> output = mindspore.ops.log2(x)
>>> print(output)
[1. 2. 3.]
```

mindspore.ops.log10

mindspore.ops.**log10**(*input*)

Compute the logarithm to the base 10 of the input tensor element-wise.

$$y_i = \log_{10}(input_i)$$

Warning: If the input value of operator log10 is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affected.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([2, 4, 10], mindspore.float16)
>>> output = mindspore.ops.log10(x)
>>> print(output)
[0.301 0.602 1.   ]
```

mindspore.ops.log1p

mindspore.ops.**log1p**(*input*)

Compute the natural logarithm of (tensor + 1) element-wise.

$$out_i = \log_e(input_i + 1)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.ops.log1p(x)
>>> print(output)
[0.6931472 1.0986123 1.609438 ]
```

mindspore.ops.logaddexp

mindspore.ops.**logaddexp**(*input*, *other*)

Computes the logarithm of the sum of exponentiations of the inputs. This function is useful in statistics where the calculated probabilities of events may be too small (exceed the range of normal floating point numbers).

$$out_i = \log(exp(input_i) + \exp(other_i))$$

Parameters

- **input** (Tensor) –The input tensor.
- **other** (Tensor) –The input tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1, 2, 3], mindspore.float16)
>>> x2 = mindspore.tensor(2, mindspore.float16)
>>> output = mindspore.ops.logaddexp(x1, x2)
>>> print(output)
[2.312 2.693 3.312]
```

mindspore.ops.logaddexp2

mindspore.ops.**logaddexp2**(*input*, *other*)

Computes the logarithm of the sum of exponentiations in base of 2 of the inputs.

$$out_i = \log_2(2^{input_i} + 2^{other_i})$$

Parameters

- **input** (Tensor) –The input tensor.
- **other** (Tensor) –The input tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([1, 2, 3], mindspore.float16)
>>> x2 = mindspore.tensor(2, mindspore.float16)
>>> output = mindspore.ops.logaddexp2(x1, x2)
>>> print(output)
[2.586 3.    3.586]
```

mindspore.ops.logical_and

mindspore.ops.**logical_and**(*input*, *other*)

Compute the "logical AND" of two tensors element-wise.

$$out_i = input_i \wedge other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, bool]*) –The first input.
- **other** (*Union[Tensor, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.ops.logical_and(x, y)
>>> print(output)
[ True False False]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_and(x, y)
>>> print(output)
False
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_and(x, y)
>>> print(output)
False
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
>>> output = mindspore.ops.logical_and(x, y)
>>> print(output)
[True False]
```

mindspore.ops.logical_not

mindspore.ops.**logical_not**(*input*)

Compute the "logical NOT" of the input tensor element-wise.

$$out_i = \neg input_i$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> output = mindspore.ops.logical_not(x)
>>> print(output)
[False  True False]
```

mindspore.ops.logical_or

mindspore.ops.**logical_or**(*input*, *other*)

Compute the "logical OR" of two tensors element-wise.

$$out_i = input_i \vee other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, bool]*) – The first input.
- **other** (*Union[Tensor, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.ops.logical_or(x, y)
>>> print(output)
[ True  True  True]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_or(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_or(x, y)
>>> print(output)
True
```

(continues on next page)

(continued from previous page)

```
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
>>> output = mindspore.ops.logical_or(x, y)
>>> print(output)
[True True]
```

mindspore.ops.logical_xor

mindspore.ops.**logical_xor**(*input*, *other*)

Compute the "logical XOR" of two tensors element-wise.

$$out_i = input_i \oplus other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.ops.logical_xor(x, y)
>>> print(output)
[False  True  True]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_xor(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.ops.logical_xor(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
```

(continues on next page)

(continued from previous page)

```
>>> output = mindspore.ops.logical_xor(x, y)
>>> print(output)
[False  True]
```

mindspore.ops.logit

mindspore.ops.**logit**(*input*, *eps=None*)

Calculate the logit of a tensor element-wise.

The formula is defined as:

$$
\begin{aligned}
y_i &= \ln(\frac{z_i}{1-z_i}) \\
z_i &= \begin{cases} input_i & \text{if eps is None} \\ \text{eps} & \text{if } input_i < \text{eps} \\ input_i & \text{if eps} \leq input_i \leq 1-\text{eps} \\ 1-\text{eps} & \text{if } input_i > 1-\text{eps} \end{cases}
\end{aligned}
$$

Parameters

- **input** (Tensor) – The input tensor.
- **eps** (*float, optional*) – The epsilon for input clamp bound. If eps is not None, the input clamp bound is defined as [eps, 1-eps], otherwise, the *input* is not clamped. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([0.1, 0.2, 0.3], mindspore.float32)
>>> output = mindspore.ops.logit(x, eps=1e-5)
>>> print(output)
[-2.1972246 -1.3862944 -0.8472978]
```

mindspore.ops.mul

mindspore.ops.**mul**(*input*, *other*)

Multiplies two tensors element-wise.

$$out_i = input_i * other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.

- Support implicit type conversion and type promotion.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – The first input.
- **other** (*Union[Tensor, number.Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> # case 1: The shape of two inputs are different
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.mul(x, 100)
>>> print(output)
[100. 200. 300.]
>>> # case 2: The shape of two inputs are the same
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> y = mindspore.tensor([4.0, 5.0, 6.0], mindspore.float32)
>>> output = mindspore.ops.mul(x, y)
>>> print(output)
[ 4. 10. 18.]
```

mindspore.ops.multiply

mindspore.ops.**multiply**(*input*, *other*)

Alias for *mindspore.ops.asinh()*.

Supported Platforms:

`Ascend` `GPU` `CPU`

mindspore.ops.mvlgamma

mindspore.ops.**mvlgamma**(*input*, *p*)

Compute the multivariate log-gamma function with dimension p element-wise. The mathematical calculation process of Mvlgamma is shown as follows:

$$\log(\Gamma_p(input)) = C + \sum_{i=1}^{p} \log(\Gamma(input - \frac{i-1}{2}))$$

where $C = \log(\pi) \times \frac{p(p-1)}{4}$ and $\Gamma(\cdot)$ is the Gamma function.

Parameters

- **input** (Tensor) –The input tensor of the multivariate log-gamma function, And the value of any element in *input* must be greater than $(p-1)/2$.
- **p** (*int*) –The number of dimensions. And the value of p must be greater than or equal to 1.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[3, 4, 5], [4, 2, 6]], mindspore.float32)
>>> y = mindspore.ops.mvlgamma(x, p=3)
>>> print(y)
[[2.694925 5.402975 9.140645]
 [5.402975 1.596312 13.64045]]
```

mindspore.ops.neg

mindspore.ops.**neg**(*input*)

Returns a tensor with negative values of the input tensor element-wise.

$$out_i = -input_i$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, -1, 2, 0, -3.5], mindspore.float32)
>>> output = mindspore.ops.neg(input)
>>> print(output)
[-1.  -2.   1.  -2.   0.   3.5]
```

mindspore.ops.negative

mindspore.ops.**negative**(*input*)

Alias for *mindspore.ops.neg()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.nextafter

mindspore.ops.**nextafter**(*input*, *other*)

Returns the next representable floating-point value after *input* towards *other* element-wise.

$$out_i = \begin{cases} input_i + eps, & \text{if } input_i < other_i \\ input_i - eps, & \text{if } input_i > other_i \\ input_i, & \text{if } input_i = other_i \end{cases}$$

Where eps is the smallest representable increment value for the input tensor's dtype.

For more detailed information, refer to A Self Regularized Non-Monotonic Neural Activation Function.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0], mindspore.float32)
>>> other = mindspore.tensor([0.1], mindspore.float32)
>>> output = mindspore.ops.nextafter(input, other)
>>> print(output)
[1.e-45]
```

mindspore.ops.polar

mindspore.ops.**polar**(*abs*, *angle*)

Converts polar coordinates to Cartesian coordinates.

Returns a complex tensor, its elements are Cartesian coordinates constructed with the polar coordinates which is specified by radial distance *abs* and polar angle *angle*.

$$y_i = abs_i * \cos(angle_i) + abs_i * \sin(angle_i) * j$$

Parameters

- **abs** (Tensor, *float*) –Radial distance.
- **angle** (Tensor, *float*) –Polar angle.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> abs = mindspore.tensor([1, 2], mindspore.float32)
>>> angle = mindspore.tensor([np.pi / 2, 5 * np.pi / 4], mindspore.float32)
>>> output = mindspore.ops.polar(abs, angle)
>>> print(output)
[ -4.3711388e-08+1.j         -1.4142137e+00-1.4142134j]
```

mindspore.ops.polygamma

mindspore.ops.**polygamma**(*n*, *input*)

Computes the *n*-th derivative of the polygamma function on *input*.

$$\psi^{(a)}(x) = \frac{d^{(a)}}{dx^{(a)}}\psi(x)$$

where $\psi(x)$ is the digamma function.

Parameters

- **n** (Tensor) –The order of the polygamma function.
- **input** (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([3.14, -2.71], mindspore.float64)
>>> a = mindspore.tensor(1, mindspore.int64)
>>> output = mindspore.ops.polygamma(a, x)
>>> print(output)
[ 0.37446456 15.49884838]
```

mindspore.ops.positive

mindspore.ops.**positive**(*input*)

Return self tensor.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([-5.0, 1.5, 3.0, 100.0], mindspore.float32)
>>> print(mindspore.ops.positive(x))
[ -5.    1.5   3.  100. ]
```

mindspore.ops.pow

mindspore.ops.**pow**(*input*, *exponent*)

Calculate the *exponent* power of each element in *input*.

Note:

- Broadcasting is supported.
- Support implicit type conversion and type promotion.

$$out_i = input_i^{exponent_i}$$

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **exponent** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.ops.pow(input, exponent=3.0)
>>> print(output)
[ 1.  8. 64.]
>>>
>>> input = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> exponent = mindspore.tensor([2.0, 4.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.pow(input, exponent)
>>> print(output)
[ 1. 16. 64.]
```

mindspore.ops.rad2deg

mindspore.ops.**rad2deg**(*x*)

Converts angles in radians to angles in degrees element-wise.

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[6.283, -3.142],[1.570, -6.283],[3.142, -1.570]], mindspore.
↪float32)
>>> output = mindspore.ops.rad2deg(x)
>>> print(output)
[[ 359.98935 -180.02333]
 [  89.95438 -359.98935]
 [ 180.02333  -89.95438]]
```

mindspore.ops.ravel

mindspore.ops.**ravel**(*input*)

Expand the multidimensional Tensor into 1D along the 0 axis direction.

Parameters

input (Tensor) – A tensor to be flattened.

Returns

Tensor, a 1-D tensor, containing the same elements of the input.

Raises

TypeError – If argument *input* is not Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.float32))
>>> output = ops.ravel(x)
>>> print(output)
[0. 1. 2. 1.]
>>> print(output.shape)
(4,)
```

mindspore.ops.real

mindspore.ops.**real**(*input*)

Return a tensor that is the real part of the input. If input is real, it is returned unchanged.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(1.3+0.4j, mindspore.complex64)
>>> output = mindspore.ops.real(input)
>>> print(output)
1.3
```

mindspore.ops.reciprocal

mindspore.ops.**reciprocal**(*input*)

Returns reciprocal of a tensor element-wise.

$$out_i = \frac{1}{x_i}$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.ops.reciprocal(input)
>>> print(output)
[1.   0.5  0.25]
```

mindspore.ops.remainder

mindspore.ops.**remainder**(*input*, *other*)

Compute the remainder of division for the input tensor element-wise.

Support implicit type conversion and type promotion.

```
remainder(input, other) == input - input.div(other, rounding_mode="floor") * other
```

Warning:

- When the elements of input exceed 2048, there might be accuracy problems.
- The calculation results of this operator on Ascend and CPU might be inconsistent.
- If shape is expressed as (D1,D2⋯,Dn), then D1*D2⋯*DN<=1000000,n<=8.

Parameters

- **input** (*Union[Tensor, numbers.Number, bool]*) –The first input.
- **other** (*Union[Tensor, numbers.Number, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> x = mindspore.tensor([-4.0, 5.0, 6.0], mindspore.float16)
>>> y = mindspore.tensor([3.0], mindspore.float16)
>>> output = mindspore.ops.remainder(x, y)
>>> print(output)
[2. 2. 0.]
>>> # case 2: The shape of two inputs are the same
>>> x = mindspore.tensor([-4.0, 5.0, 6.0], mindspore.float16)
>>> y = mindspore.tensor([3.0, 2.0, 3.0], mindspore.float16)
>>> output = mindspore.ops.remainder(x, y)
>>> print(output)
[2. 1. 0.]
```

mindspore.ops.rot90

`mindspore.ops.`**`rot90`**(*input*, *k*, *dims*)

Rotate a n-D tensor by 90 degrees in the plane specified by dims axis. Rotation direction is from the first towards the second axis if k > 0, and from the second towards the first for k < 0.

Parameters

- **input** (`Tensor`) –The input tensor.
- **k** (*int*) –Number of times to rotate.
- **dims** (*Union[list(int), tuple(int)]*) –Axis to rotate.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 2, 3],
...                       [4, 5, 6]], mindspore.float32)
>>> output = mindspore.ops.rot90(x, k=1, dims=(0, 1))
>>> print(output)
[[3. 6.]
 [2. 5.]
 [1. 4.]]
>>> mindspore.ops.rot90(x, k=1, dims=(1, 0)) == mindspore.ops.rot90(x, k=-1, dims=(0,1))
Tensor(shape=[3, 2], dtype=Bool, value=
[[ True,  True],
 [ True,  True],
 [ True,  True]])
>>> # when input array has ndim>2
>>> x = mindspore.tensor([[[1, 2, 3],
...                        [4, 5, 6]],
...                       [[7, 8, 9],
...                        [10, 11, 12]]], mindspore.float32)
>>> output = mindspore.ops.rot90(x, k=1, dims=(2, 1))
>>> print(output)
[[[ 4.  1.]
  [ 5.  2.]
  [ 6.  3.]]
 [[10.  7.]
  [11.  8.]
  [12.  9.]]]
```

mindspore.ops.round

mindspore.ops.**round**(*input*, *, *decimals=0*)

Round elements of input to the nearest integer.

$$out_i \approx input_i$$

Note: The input data types supported by the Ascend platform include bfloat16 (Atlas training series products are not supported), float16, float32, float64, int32, and int64.

Parameters

input (Tensor) −The input tensor.

Keyword Arguments

decimals (int, *optional*) −Number of decimal places to round. If decimals is negative, it specifies the number of positions to the left of the decimal point. Default 0 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.round(mindspore.tensor([4.7, -2.3, 9.1, -7.7]))
>>> print(output)
[ 5. -2.  9. -8.]
>>> # Values equidistant from two integers are rounded towards the
>>> # the nearest even value (zero is treated as even)
>>> output = mindspore.ops.round(mindspore.tensor([-0.5, 0.5, 1.5, 2.5]))
>>> print(output)
[0. 0.  2.  2.]
>>> # A positive decimals argument rounds to the to that decimal place
>>> output = mindspore.ops.round(mindspore.tensor([0.1234567]), decimals=3)
>>> print(output)
[0.123]
>>> # A negative decimals argument rounds to the left of the decimal
>>> output = mindspore.ops.round(mindspore.tensor([1200.1234567]), decimals=-3)
>>> print(output)
[1000.]
```

mindspore.ops.rsqrt

mindspore.ops.**rsqrt**(*input*)

Compute reciprocal of square root of input tensor element-wise.

$$out_i = \frac{1}{\sqrt{input_i}}$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-0.0370,  0.2970,  1.5420, -0.9105])
>>> output = mindspore.ops.rsqrt(input)
>>> print(output)
[       nan 1.8349396  0.8053002        nan]
```

mindspore.ops.sgn

mindspore.ops.**sgn**(*input*)

Extension of *mindspore.ops.sign()* in complex domain. For real number input, this function is the same as *mindspore.ops.sign()*. For complex input, this function is calculated according to the following formula.

$$\text{out}_i = \begin{cases} 0 & |\text{input}_i| == 0 \\ \frac{\text{input}_i}{|\text{input}_i|} & \text{otherwise} \end{cases}$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[3+4j, 7-24j, 0, 6+8j, 8], [15+20j, 7-24j, 0, 3+4j, 20]],␣
↪mindspore.complex64)
>>> output = mindspore.ops.sgn(input)
>>> print(output)
[[0.6 +0.8j  0.28-0.96j 0.  +0.j   0.6 +0.8j  1.  +0.j  ]
 [0.6 +0.8j  0.28-0.96j 0.  +0.j   0.6 +0.8j  1.  +0.j  ]]
```

mindspore.ops.sign

mindspore.ops.**sign**(*input*)

Return an element-wise indication of the sign of a number.

$$\text{out}_i = \begin{cases} -1 & \text{input}_i < 0 \\ 0 & \text{input}_i = 0 \\ 1 & \text{input}_i > 0 \end{cases}$$

Note: When the input is NaN and dtype is float64, the output of this operator is NaN.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-1, 0, 2, 4, 6], [2, 3, 5, -6, 0]])
>>> output = mindspore.ops.sign(input)
>>> print(output)
[[-1  0  1  1  1]
 [ 1  1  1 -1  0]]
>>> mindspore.set_device(device_target="CPU")
>>> x = mindspore.tensor([[-1, 0, float('inf'), 4, float('nan')], [2, 3, float('-inf'), -6,␣
↪0]])
>>> output = mindspore.ops.sign(x)
>>> print(output)
[[-1.  0.  1.  1.  0.]
 [ 1.  1. -1. -1.  0.]]
```

mindspore.ops.signbit

mindspore.ops.**signbit**(*input*)

Determine the symbol of each element. If the element value is less than 0, the corresponding output position is True; otherwise, it is False.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.3, 1.2, 0., -2.5])
>>> output = mindspore.ops.signbit(input)
>>> print(output)
[False False False  True]
```

mindspore.ops.sin

mindspore.ops.**sin**(*input*)

Compute sine of the input tensor element-wise.

$$output_i = \sin(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.ops.sin(input)
>>> print(output)
[0.58103514 0.27635565 0.4168708 0.58103514]
```

mindspore.ops.sinc

mindspore.ops.**sinc**(*input*)

Compute the normalized sinc of input.

$$out_i = \begin{cases} \frac{sin(\pi input_i)}{\pi input_i} & input_i \neq 0 \\ 1 & input_i = 0 \end{cases}$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.ops.sinc(input)
>>> print(output)
[0.47735003 0.8759357  0.7224278  0.47735003]
```

mindspore.ops.sinh

mindspore.ops.**sinh**(*input*)

Compute hyperbolic sine of the input element-wise.

$$output_i = \sinh(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.ops.sinh(input)
>>> print(output)
[0.6604918  0.28367308 0.44337422 0.6604918 ]
```

mindspore.ops.sqrt

mindspore.ops.**sqrt**(*x*)

Return sqrt of a tensor element-wise.

$$out_i = \sqrt{x_i}$$

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 4.0, 9.0], mindspore.float32)
>>> output = mindspore.ops.sqrt(x)
>>> print(output)
[1. 2. 3.]
```

mindspore.ops.square

mindspore.ops.**square**(*input*)

Return square of a tensor element-wise.

$$y_i = input_i^2$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> output = mindspore.ops.square(input)
>>> print(output)
[1. 4. 9.]
```

mindspore.ops.sub

mindspore.ops.**sub**(*input*, *other*)

Subtract the second input from the first input element-wise.

$$out_i = input_i - other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- Support implicit type conversion and type promotion.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – The first input.
- **other** (*Union[Tensor, number.Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([4, 5, 6], mindspore.int32)
>>> output = mindspore.ops.sub(input, other)
>>> print(output)
[-3 -3 -3]
```

mindspore.ops.subtract

mindspore.ops.**subtract**(*input*, *other*, *, *alpha=1*)

Subtract *other* scaled by *alpha* from *input*.

Support implicit type conversion and type promotion.

$$output[i] = input[i] - alpha * other[i]$$

Parameters

- **input** (*Union[Tensor, number.Number]*) –The first input.
- **other** (*Union[Tensor, number.Number]*) –The second input.

Keyword Arguments

alpha (*number*) –The multiplier for *other*. Default `1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([4, 5, 6], mindspore.float32)
>>> y = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> z = mindspore.ops.subtract(input, y, alpha=1)
>>> print(z)
[3. 3. 3.]
```

mindspore.ops.t

mindspore.ops.**t**(*input*)

Transpose a 2-D tensor. 0-D and 1-D tensor are returned as it is.

Parameters

input (`Tensor`) –The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: Input is a 0-D tensor
>>> mindspore.ops.t(mindspore.tensor(6))
Tensor(shape=[], dtype=Int64, value= 6)
>>>
>>> # case 2: Input is a 1-D tensor
>>> mindspore.ops.t(mindspore.tensor([1, 2]))
Tensor(shape=[2], dtype=Int64, value= [1, 2])
>>>
>>> # case 3: Input is a 2-D tensor
>>> mindspore.ops.t(mindspore.tensor([[1, 2, 3], [2, 3, 4]]))
Tensor(shape=[3, 2], dtype=Int64, value=
[[1, 2],
 [2, 3],
 [3, 4]])
```

mindspore.ops.tan

mindspore.ops.**tan**(*input*)

Computes tangent of *input* element-wise.

$$out_i = \tan(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.tan(mindspore.tensor([-1.0, 0.0, 1.0]))
>>> print(output)
[-1.5574077 0. 1.5574077]
```

mindspore.ops.tanhshrink

mindspore.ops.**tanhshrink**(*input*)

Apply the element-wise Tanhshrink function.

$$Tanhshrink(x) = x - Tanh(x)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> output = mindspore.ops.tanhshrink(mindspore.tensor([1., 2., 3., 2., 1.]))
>>> print(output)
[0.23840582 1.0359724  2.0049443  1.0359724  0.23840582]
```

mindspore.ops.trapz

`mindspore.ops.`**`trapz`**(*y*, *x=None*, ***, *dx=1.0*, *dim=-1*)

Compute the trapezoidal rule along dim.

The spacing between elements is specified by the tensor *x* or the scalar *dx* , default `1` .

$$\int y(x)\mathrm{d}x$$

Parameters

- **y** (`Tensor`) – The input tensor to integrate.
- **x** (`Tensor, optional`) – If specified, defines spacing between values.

Keyword Arguments

- **dx** (`float, optional`) – The spacing between elements. Default `1.0` . If *x* is specified, *dx* does not take effect.
- **dim** (`int, optional`) – The dimension along which to compute the trapezoidal rule. Default `-1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> y = mindspore.tensor([[1., 2., 3.], [2., 3., 4.], [3., 2., 1.]])
>>> # case 1: Integrate over a regular grid, with spacing 1.
>>> output = mindspore.ops.trapz(y, dx=1.)
>>> print(output)
[4. 6. 4.]
>>>
>>> # case 2: Integrate over an irregular grid.
>>> x = mindspore.tensor([[1, 2, 3], [1, 3, 5], [1, 4, 7]])
>>> output = mindspore.ops.trapz(y, x)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[ 4. 12. 12.]
```

mindspore.ops.tril_indices

mindspore.ops.**tril_indices**(*row*, *col*, *offset=0*, *, *dtype=mstype.int64*)

Return a 2-by-N tensor containing the indices of the lower triangular elements of a *row* * *col* matrix. The first row of the Tensor contains row coordinates, and the second row contains column coordinates. The coordinates are sorted by rows and then columns.

Note: When running on CUDA, row * col must be less than 2^59 to prevent overflow during calculation.

Parameters

- **row** (*int*) −number of rows in the 2-D matrix.
- **col** (*int*) −number of columns in the 2-D matrix.
- **offset** (*int, optional*) −Diagonal offset. Default `0` .
  - When *offset* is a positive integer, the diagonal is shifted upward.
  - When *offset* is a negative integer, the diagonal is shifted downward.

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The specified type of output tensor. An optional data type of *mindspore.int32* and *mindspore.int64*. Default `mstype.int64` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: By default, offset=0, all elements on and below the main diagonal are retained.
>>> mindspore.ops.tril_indices(3, 2, 0)
Tensor(shape=[2, 5], dtype=Int64, value=
[[0, 1, 1, 2, 2],
 [0, 0, 1, 0, 1]])
>>>
>>> # case 2: Offset=1, the indices on and below the first sub-diagonal above the main␣
↪diagonal are returned.
>>> mindspore.ops.tril_indices(3, 2, 1)
Tensor(shape=[2, 6], dtype=Int64, value=
[[0, 0, 1, 1, 2, 2],
 [0, 1, 0, 1, 0, 1]])
>>>
>>> # case 3: Offset=-1, the indices on and below the first sub-diagonal below the main␣
↪diagonal are returned.
>>> mindspore.ops.tril_indices(3, 2, -1)
```

(continues on next page)

(continued from previous page)

```
Tensor(shape=[2, 3], dtype=Int64, value=
[[1, 2, 2],
 [0, 0, 1]])
```

mindspore.ops.triu_indices

mindspore.ops.**triu_indices**(*row*, *col*, *offset=0*, *, *dtype=mstype.int64*)

Return a 2-by-N tensor containing the indices of the upper triangular elements of a *row* * *col* matrix. The first row of the Tensor contains row coordinates, and the second row contains column coordinates. The coordinates are sorted by rows and then columns.

Note: When running on CUDA, row * col must be less than 2^59 to prevent overflow during calculation.

Parameters

- **row** (*int*) −number of rows in the 2-D matrix.
- **col** (*int*) −number of columns in the 2-D matrix.
- **offset** (*int, optional*) −Diagonal offset. Default `0` .
  - When *offset* is a positive integer, the diagonal is shifted upward.
  - When *offset* is a negative integer, the diagonal is shifted downward.

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The specified type of output tensor. An optional data type of *mindspore.int32* and *mindspore.int64*. Default `mstype.int64` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: By default, offset=0, all elements on and below the main diagonal are retained.
>>> mindspore.ops.triu_indices(3, 2, 0)
Tensor(shape=[2, 3], dtype=Int64, value=
[[0, 0, 1],
 [0, 1, 1]])
>>>
>>> # case 2: If offset=-1, the indices on and above the first sub-diagonal below the main␣
 ↪diagonal are returned.
>>> mindspore.ops.triu_indices(3, 2, -1)
Tensor(shape=[2, 5], dtype=Int64, value=
[[0, 0, 1, 1, 2],
 [0, 1, 0, 1, 1]])
```

mindspore.ops.true_divide

mindspore.ops.**true_divide**(*dividend*, *divisor*)

Alias for *mindspore.ops.div()* with rounding_mode=None .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.trunc

mindspore.ops.**trunc**(*input*)

Returns a tensor with the truncated integer values of the elements of the input tensor.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.trunc(mindspore.tensor([3.4742, 0.5466, -0.8008, -3.9079]))
>>> print(output)
[3. 0. 0. -3.]
```

mindspore.ops.truncate_div

mindspore.ops.**truncate_div**(*x*, *y*)

Divide the first input tensor *x* by the second input tensor *y* element-wise and rounds the results of division towards zero.

Note: Support implicit type conversion and broadcasting.

Parameters

- **x** (*Union[Tensor, Number, bool]*) –The first input tensor.
- **y** (*Union[Tensor, Number, bool]*) –The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.truncate_div(mindspore.tensor([2, 4, -1]), mindspore.tensor([3, 3, 3]))
Tensor(shape=[3], dtype=Int64, value= [0, 1, 0])
```

mindspore.ops.truncate_mod

mindspore.ops.**truncate_mod**(*x*, *y*)

Return the remainder of division element-wise.

Support implicit type conversion and broadcasting.

Warning:

- The input data does not support 0.
- When the elements of input exceed 2048 , the accuracy of operator cannot guarantee the requirement of double thousandths in the mini form.
- Due to different architectures, the calculation results of this operator on NPU and CPU may be inconsistent.
- If shape is expressed as (D1,D2⋯,Dn), then D1*D2⋯*DN<=1000000,n<=8.

Parameters

- **x** (*Union[Tensor, Number, bool]*) – The first input tensor.
- **y** (*Union[Tensor, Number, bool]*) – The second input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> output = mindspore.ops.truncate_mod(mindspore.tensor([2., 4., -1.]), mindspore.tensor([3.
↪, 3., 3.]))
>>> print(output)
[ 2.  1. -1.]
```

mindspore.ops.xdivy

mindspore.ops.**xdivy**(*x*, *y*)

Divide *x* by *y* element-wise.

Note:

- Support broadcast, support implicit type conversion and type promotion.
- When *x* and *y* are both of datatype complex, they should be both complex64 or complex128 at the same time.
- *x* and *y* can not be both bool at the same time.

Parameters

- **x** (*Union[Tensor, Number, bool]*) – Numerator tensor.
- **y** (*Union[Tensor, Number, bool]*) – Denominator tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.xdivy(mindspore.tensor([2., 4., -1.]), mindspore.tensor([2., 2.,␣
↪2.]))
>>> print(output)
[ 1.   2.  -0.5]
```

mindspore.ops.xlogy

mindspore.ops.**xlogy**(*input*, *other*)

Compute *input* multiplied by the logarithm of *other* element-wise.

$$out_i = input_i * lnother_i$$

Note:

- Support broadcast, support implicit type conversion and type promotion.

Warning: On Ascend, the data type of *input* and *other* must be float16 or float32.

Parameters

- **input** (*Union[Tensor, numbers.Number, bool]*) – The first input tensor.

- **other** (*Union[Tensor, numbers.Number, bool]*) – The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.xlogy(mindspore.tensor([-5., 0., 4.]), mindspore.tensor([2., 2.,␣
↪2.]))
>>> print(output)
[-3.465736   0.         2.7725887]
```

mindspore.ops.zeta

mindspore.ops.**zeta**(*input*, *other*)

Elemental-wise compute the Hurwitz zeta function values.

$$\zeta(x, q) = \sum_{k=0}^{\infty} \frac{1}{(k+q)^x}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (*Union[Tensor, int, float]*) – The first input tensor. Represented as x in the formula.
- **other** (*Union[Tensor, int, float]*) – The second input tensor. Represented as q in the formula.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> z = mindspore.ops.zeta(mindspore.tensor([10.]), mindspore.tensor([1.]))
>>> print(z)
[1.0009946]
```

2.2.2 Reduction Functions

API Name	Description	Supported Platforms
mindspore.ops.all	Tests if all element in *input* evaluates to *True* along the given axes.	Ascend GPU CPU
mindspore.ops.amax	Return the maximum values along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.amin	Return the minimum values along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.aminmax	Return the minimum values and maximum values along the given axes of the tensor.	Ascend GPU CPU
mindspore.ops.any	Tests if any element in *input* evaluates to *True* along the given axes.	Ascend GPU CPU
mindspore.ops.argmax	Return the indices of the maximum values along a specified dimension of the tensor.	Ascend GPU CPU
mindspore.ops.argmin	Return the indices of the minimum values along a specified dimension of the tensor.	Ascend GPU CPU
mindspore.ops.cummax	Return the cumulative maximum values and their indices along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.cummin	Return the cumulative minimum values and their indices along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.cumprod	Return the cumulative product along the given dimension of the tensor.	Ascend GPU CPU
mindspore.ops.cumsum	Return the cumulative sum along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.histc	Computes the histogram of a tensor.	Ascend CPU
mindspore.ops.logcumsumexp	Calculate the log of cumulative summed exponentials of the tensor along a specified dimension.	Ascend CPU GPU
mindspore.ops.logsumexp	Calculate the log of summed exponentials of the tensor along a specified dimension.	Ascend GPU CPU
mindspore.ops.max	Return the maximum values and their indices along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.mean	Compute the mean(s) of the tensor along the specified axis(axes).	Ascend GPU CPU
mindspore.ops.median	Return the median(s) and indice(s) of the tensor along the specified axis.	GPU CPU
mindspore.ops.min	Return the minimum values and their indices along the given axis of the tensor.	Ascend GPU CPU
mindspore.ops.norm	Compute the matrix norm or vector norm of the tensor along a specified dimension.	Ascend GPU CPU
mindspore.ops.prod	Return the product(s) of the tensor along the specified axis(axes).	Ascend GPU CPU
mindspore.ops.std	Compute the standard deviation of the tensor along a specified axis.	Ascend GPU CPU
mindspore.ops.std_mean	Compute the standard deviation and the mean of the tensor along a specified axis.	Ascend GPU CPU
mindspore.ops.var	Compute the variance of the tensor along a specified axis.	Ascend GPU CPU
mindspore.ops.var_mean	Compute the variance and the mean of the tensor along a specified axis.	Ascend GPU CPU

mindspore.ops.all

mindspore.ops.**all**(*input*, *axis=None*, *keep_dims=False*)

Tests if all element in *input* evaluates to *True* along the given axes.

Parameters

- **input** (Tensor) –The input Tensor.
- **axis** (*Union[int, tuple(int), list(int), Tensor], optional*) –The dimensions to reduce. If None , all dimensions are reduced. Default None .
- **keep_dims** (*bool, optional*) –Whether the output tensor has dim retained or not. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.ops.all tests along all the axes.
>>> mindspore.ops.all(input)
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 2: Reduces a dimension along axis 1, with keep_dims False.
>>> mindspore.ops.all(input, axis=1)
Tensor(shape=[2], dtype=Bool, value= [False,  True])
>>>
>>> # case 3: Reduces a dimension along axis (0,1), with keep_dims False.
>>> mindspore.ops.all(input, axis=(0,1))
Tensor(shape=[], dtype=Bool, value= False)
>>>
>>> # case 4: Reduces a dimension along axis [0,1], with keep_dims True.
>>> mindspore.ops.all(input, axis=[0,1], keep_dims=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[False]])
```

mindspore.ops.amax

mindspore.ops.**amax**(*input*, *axis=None*, *keepdims=False*, ***, *initial=None*, *where=None*)

Return the maximum values along the given axis of the tensor.

Parameters

- **input** (Tensor*[Number]*) –The input tensor.
- **axis** (*Union[int, tuple(int), list(int), Tensor], optional*) –Specify the axis for computation. If None , compute all elements in the *input* . Default None .
- **keepdims** (*bool, optional*) –Whether the output tensor has dim retained. Default False .

Keyword Arguments

- **initial** (*scalar, optional*) –Initial value for the maximum. Default `None` .
- **where** (`Tensor[bool], optional`) –Specifies the range over which to compute the maximum values. The shape of this tensor must be broadcastable to the shape of *input* . An *initial* value must be specified. Default `None` , indicating that all elements are to be computed.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum of all elements.
>>> mindspore.ops.amax(input)
Tensor(shape=[], dtype=Int64, value= 9)
>>>
>>> # case 2: Compute maximum along axis 1.
>>> mindspore.ops.amax(input, axis=1)
Tensor(shape=[3], dtype=Int64, value= [9, 7, 8])
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.amax(input, axis=1, keepdims=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[9],
 [7],
 [8]])
>>>
>>> # case 4: Use "where" to include only specific elements in computing the maximum.
>>> where = mindspore.tensor([[0, 0, 1, 0],
...                           [0, 0, 1, 1],
...                           [1, 1, 1, 0]], dtype=mindspore.bool_)
>>> mindspore.ops.amax(input, axis=1, keepdims=True, initial=0, where=where)
Tensor(shape=[3, 1], dtype=Int64, value=
[[4],
 [7],
 [8]])
>>>
>>> # case 5: The shape of "where" must be broadcast compatible with input.
>>> where = mindspore.tensor([[False],
...                           [False],
...                           [False]])
>>> mindspore.ops.amax(input, axis=0, keepdims=True, initial=0, where=where)
Tensor(shape=[1, 4], dtype=Int64, value=
[[0, 0, 0, 0]])
```

mindspore.ops.amin

mindspore.ops.**amin**(*input*, *axis=None*, *keepdims=False*, *, *initial=None*, *where=None*)

Return the minimum values along the given axis of the tensor.

Parameters

- **input** (Tensor[Number]) –The input tensor.
- **axis** (Union[int, tuple(int), list(int), Tensor], optional) –Specify the axis for computation. If None , compute all elements in the *input* . Default None .
- **keepdims** (bool, optional) –Whether the output tensor has dim retained. Default False .

Keyword Arguments

- **initial** (scalar, optional) –Initial value for the minimum. Default None .
- **where** (Tensor[bool], optional) –Specifies the range over which to compute the minimum values. The shape of this tensor must bebroadcastable to the shape of *input* . An *initial* value must be specified. Default None , indicating that all elements are to be computed.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[2, 5, 1, 6],
...                           [3, -7, -2, 4],
...                           [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum of all elements.
>>> mindspore.ops.amin(input)
Tensor(shape=[], dtype=Int64, value= -7)
>>>
>>> # case 2: Compute minimum along axis 1.
>>> mindspore.ops.amin(input, axis=1)
Tensor(shape=[3], dtype=Int64, value= [ 1, -7, -4])
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.amin(input, axis=1, keepdims=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[ 1],
 [-7],
 [-4]])
>>>
>>> # case 4: Use "where" to include only specific elements in computing the minimum.
>>> where = mindspore.tensor([[1, 0, 1, 0],
...                           [0, 0, 1, 1],
...                           [1, 1, 1, 0]], dtype=mindspore.bool_)
>>> mindspore.ops.amin(input, axis=1, keepdims=True, initial=0, where=where)
Tensor(shape=[3, 1], dtype=Int64, value=
 [[ 0],
  [-2],
```

(continues on next page)

(continued from previous page)

```
  [-4]])
>>>
>>> # case 5: The shape of "where" must be broadcast compatible with input.
>>> where = mindspore.tensor([[False],
...                           [False],
...                           [False]])
>>> mindspore.ops.amin(input, axis=0, keepdims=True, initial=0, where=where)
Tensor(shape=[1, 4], dtype=Int64, value=
 [[0, 0, 0, 0]])
```

mindspore.ops.aminmax

mindspore.ops.**aminmax**(*input*, ***, *axis=0*, *keepdims=False*)

Return the minimum values and maximum values along the given axes of the tensor.

Parameters

input (Tensor) – The input tensor.

Keyword Arguments

- **axis** (int, *optional*) – Specify the axis for computation. If None , compute all elements in the *input* . Default 0 .
- **keepdims** (bool, *optional*) – Whether the output tensor has dim retained. Default False .

Returns

Tuple(min, max) of 2 tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>>
>>> # case 1: By default, compute along axis 0.
>>> mindspore.ops.aminmax(input)
(Tensor(shape=[4], dtype=Int64, value= [5, 1, 3, 4]),
 Tensor(shape=[4], dtype=Int64, value= [9, 3, 7, 6]))
>>>
>>> # case 2: Disregard NaN (Not a Number) values present in the input during computation.
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, float('nan')]])
>>> mindspore.ops.aminmax(input, axis=None)
(Tensor(shape=[], dtype=Float32, value= 1),
 Tensor(shape=[], dtype=Float32, value= 9))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.aminmax(input, axis=None, keepdims=True)
```

(continues on next page)

(continued from previous page)

```
(Tensor(shape=[1, 1], dtype=Float32, value=
 [[ 1.00000000e+00]]),
 Tensor(shape=[1, 1], dtype=Float32, value=
 [[ 9.00000000e+00]]))
```

mindspore.ops.any

mindspore.ops.**any**(*input*, *axis=None*, *keep_dims=False*)

Tests if any element in *input* evaluates to *True* along the given axes.

Parameters

- **input** (Tensor) – The input tensor.
- **axis** (*Union[int, tuple(int), list(int), Tensor], optional*) – The dimensions to reduce. If None , all dimensions are reduced. Default None .
- **keep_dims** (*bool, optional*) – Whether the output tensor has dim retained or not. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.ops.any tests along all the axes.
>>> mindspore.ops.any(input)
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 2: Reduces a dimension along axis 1, with keep_dims False.
>>> mindspore.ops.any(input, axis=1)
Tensor(shape=[2], dtype=Bool, value= [ True,  True])
>>>
>>> # case 3: Reduces a dimension along axis (0, 1), with keep_dims False.
>>> mindspore.ops.any(input, axis=(0,1))
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 4: Reduces a dimension along axis [0, 1], with keep_dims True.
>>> mindspore.ops.any(input, axis=[0,1], keep_dims=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[ True]])
```

mindspore.ops.argmax

mindspore.ops.**argmax**(*input*, *dim=None*, *keepdim=False*)

Return the indices of the maximum values along a specified dimension of the tensor.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*Union[int, None], optional*) –Specify the dimension for computation. If None , compute maximum value indexes of all elements in the *input* . Default None .
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained. Default False .

Returns

Tensor, contains the index of the maximum value.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum indice of all elements.
>>> mindspore.ops.argmax(input)
Tensor(shape=[], dtype=Int64, value= 0)
>>>
>>> # case 2: Compute maximum indice along dim 1.
>>> mindspore.ops.argmax(input, dim=1)
Tensor(shape=[3], dtype=Int64, value= [0, 2, 0])
>>>
>>> # case 3: If keepdim=True, the output shape will be same of that of the input.
>>> mindspore.ops.argmax(input, dim=1, keepdim=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[0],
 [2],
 [0]])
```

mindspore.ops.argmin

mindspore.ops.**argmin**(*input*, *axis=None*, *keepdims=False*)

Return the indices of the minimum values along a specified dimension of the tensor.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*Union[int, None], optional*) –Specify the dimension for computation. If None , compute all elements in the *input* . Default None .
- **keepdims** (*bool, optional*) –Whether the output tensor has dim retained. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[2, 5, 1, 6],
...                           [3, -7, -2, 4],
...                           [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum indice of all elements.
>>> mindspore.ops.argmin(input)
Tensor(shape=[], dtype=Int32, value= 5)
>>>
>>> # case 2: Compute the minimum indices along axis 1.
>>> mindspore.ops.argmin(input, axis=1)
Tensor(shape=[3], dtype=Int32, value= [2, 1, 1])
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.argmin(input, axis=1, keepdims=True)
Tensor(shape=[3, 1], dtype=Int32, value=
[[2],
 [1],
 [1]])
```

mindspore.ops.cummax

mindspore.ops.**cummax**(*input*, *axis*)

Return the cumulative maximum values and their indices along the given axis of the tensor.

$$y_i = \max(x_1, x_2, ..., x_i)$$

Parameters

- **input** (Tensor) −The input tensor.
- **axis** (*int*) −Specify the axis for computation.

Returns

Tuple(max, max_indices) of 2 tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[3, 4, 6, 10],
...                           [1, 6, 7, 9],
...                           [4, 3, 8, 7],
...                           [1, 3, 7, 9]])
>>> mindspore.ops.cummax(input, axis=0)
(Tensor(shape=[4, 4], dtype=Int64, value=
 [[ 3,  4,  6, 10],
  [ 3,  6,  7, 10],
  [ 4,  6,  8, 10],
  [ 4,  6,  8, 10]]),
 Tensor(shape=[4, 4], dtype=Int64, value=
 [[0, 0, 0, 0],
  [0, 1, 1, 0],
  [2, 1, 2, 0],
  [2, 1, 2, 0]]))
```

mindspore.ops.cummin

mindspore.ops.**cummin**(*input*, *axis*)

Return the cumulative minimum values and their indices along the given axis of the tensor.

$$y_i = \min(x_1, x_2, ..., x_i)$$

Parameters

- **input** (Tensor) −The input tensor.
- **axis** (*int*) −Specify the axis for computation.

Returns

Tuple(min, min_indices) of 2 tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor, ops
>>> import mindspore
>>> a = Tensor([-0.2284, -0.6628,  0.0975,  0.2680, -1.3298, -0.4220], mindspore.float32)
>>> output = ops.cummin(a, axis=0)
>>> print(output[0])
[-0.2284 -0.6628 -0.6628 -0.6628 -1.3298 -1.3298]
>>> print(output[1])
[0 1 1 1 4 4]
```

mindspore.ops.cumprod

mindspore.ops.**cumprod**(*input*, *dim*, *dtype=None*)

Return the cumulative product along the given dimension of the tensor.

$$y_i = x_1 * x_2 * x_3 * ... * x_i$$

Parameters

- **input** (Tensor) −The input tensor.
- **dim** (*int*) −Specify the dimension for computation.
- **dtype** (*mindspore.dtype*, optional) −The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3],
...                           [4, 5, 6]])
>>> mindspore.ops.cumprod(input, dim=0)
Tensor(shape=[2, 3], dtype=Int64, value=
[[ 1,  2,  3],
 [ 4, 10, 18]])
>>> mindspore.ops.cumprod(input, dim=1)
Tensor(shape=[2, 3], dtype=Int64, value=
[[  1,   2,   6],
 [  4,  20, 120]])
```

mindspore.ops.cumsum

mindspore.ops.**cumsum**(*x*, *axis*, *dtype=None*)

Return the cumulative sum along the given axis of the tensor.

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Parameters

- **x** (Tensor) −The input tensor.
- **axis** (*int*) −Specify the axis for computation.
- **dtype** (*mindspore.dtype*, optional) −The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3],
...                           [4, 5, 6]], mindspore.int32)
>>> mindspore.ops.cumsum(input, axis=0)
Tensor(shape=[2, 3], dtype=Int32, value=
[[1, 2, 3],
 [5, 7, 9]])
>>> mindspore.ops.cumsum(input, axis=1)
Tensor(shape=[2, 3], dtype=Int32, value=
[[ 1,  3,  6],
 [ 4,  9, 15]])
```

mindspore.ops.histc

mindspore.ops.**histc**(*input*, *bins=100*, *min=0.*, *max=0.*)

Computes the histogram of a tensor.

The elements are sorted into equal width bins between *min* and *max*. If *min* and *max* are both zero, the minimum and maximum values of the data are used.

Elements lower than min or higher than max are ignored.

Parameters

- **input** (Tensor) –The input tensor.
- **bins** (*int, optional*) –Number of histogram bins. Default 100 .
- **min** (*int, float, optional*) –Minimum value of the histogram data range. Default 0. .
- **max** (*int, float, optional*) –Maximum value of the histogram data range. Default 0. .

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1., 1, 2, 2, 2, 5, 8])
>>> output = mindspore.ops.histc(input, bins=5, min=0, max=9)
>>> print(output)
[2. 3. 1. 0. 1.]
```

mindspore.ops.logcumsumexp

mindspore.ops.**logcumsumexp**(*input*, *axis*)

Calculate the log of cumulative summed exponentials of the tensor along a specified dimension.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **axis** (*int*) –Specify the axis for computation.

Returns

Tensor

Supported Platforms:

Ascend CPU GPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9., 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: Compute the log of cumulative summed exponential along dim 0.
>>> output = mindspore.ops.logcumsumexp(input, 0)
>>> print(output)
[[9.        3.        4.        5.       ]
 [9.01815   3.3132617 7.0485873 5.3132615]
 [9.326563  3.407606  7.065884  6.407606 ]]
>>>
>>> # case 2: Compute the log of cumulative summed exponential along dim 1.
>>> output = mindspore.ops.logcumsumexp(input, 1)
>>> print(output)
[[9.        9.002476  9.009174  9.02716  ]
 [5.        5.0485873 7.1328454 7.175515 ]
 [8.        8.000912  8.007621  8.133643 ]]
```

mindspore.ops.logsumexp

mindspore.ops.**logsumexp**(*input*, *dim*, *keepdim=False*)

Calculate the log of summed exponentials of the tensor along a specified dimension.

$$logsumexp(input) = \log(\sum(e^{input-input_{max}})) + input_{max}$$

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*Union[int, tuple(int), list(int)]*) –Specify the dimension for computation. If `()` , compute all elements in the *input* .
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9., 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the log of summed exponential of all elements.
>>> output = mindspore.ops.logsumexp(input, ())
>>> print(output)
9.475807
>>>
>>> # case 2: Compute the log of summed exponential along dim 0.
>>> output = mindspore.ops.logsumexp(input, 0)
>>> print(output)
[9.326562  3.4076054 7.065884  6.4076056]
>>>
>>> # case 3: If keepdim=True, the output shape will be same of that of the input.
>>> output = mindspore.ops.logsumexp(input, 1, True)
>>> print(output)
[[9.02716 ]
 [7.175515]
 [8.133643]]
```

mindspore.ops.max

`mindspore.ops.`**`max`**(*input*, *axis=None*, *keepdims=False*, ***, *initial=None*, *where=None*)

Return the maximum values and their indices along the given axis of the tensor.

Parameters

- **`input`** (`Tensor`) −The input tensor.
- **`axis`** (*`int`*) −Specify the axis for computation. If `None` , compute all elements in the *input* . Default: `False` .
- **`keepdims`** (*`bool`*) −Whether the output tensor has dim retained. Default `False`.

Keyword Arguments

- **`initial`** (*`scalar, optional`*) −Initial value for the maximum. Default `None`.
- **`where`** (`Tensor`*`[bool], optional`*) −Specifies the range over which to compute the maximum values. The shape of this tensor must be broadcastable to the shape of *input* . An *initial* value must be specified. Default `None` , indicating that all elements are to be computed.

Returns

Tuple(max, max_indices) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the maximum of all elements.
>>> mindspore.ops.max(input)
(Tensor(shape=[], dtype=Int64, value= 9),
 Tensor(shape=[], dtype=Int64, value= 0))
>>>
>>> # case 2: Compute maximum along axis 1.
>>> mindspore.ops.max(input, axis=1)
(Tensor(shape=[3], dtype=Int64, value= [9, 7, 8]),
 Tensor(shape=[3], dtype=Int64, value= [0, 2, 0]))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.max(input, axis=1, keepdims=True)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[9],
  [7],
  [8]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[0],
  [2],
  [0]]))
>>>
>>> # case 4: Use "where" to include only specific elements in computing the maximum.
>>> where = mindspore.tensor([[0, 0, 1, 0],
...                           [0, 0, 1, 1],
...                           [1, 1, 1, 0]], dtype=mindspore.bool_)
>>> mindspore.ops.max(input, axis=1, keepdims=True, initial=0, where=where)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[4],
  [7],
  [8]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[2],
  [2],
  [0]]))
>>>
>>> # case 5: The shape of "where" must be broadcast compatible with input.
>>> where = mindspore.tensor([[False],
...                           [False],
...                           [False]])
>>> mindspore.ops.max(input, axis=0, keepdims=True, initial=0, where=where)
(Tensor(shape=[1, 4], dtype=Int64, value=
 [[0, 0, 0, 0]]),
 Tensor(shape=[1, 4], dtype=Int64, value=
 [[0, 0, 0, 0]]))
```

mindspore.ops.mean

mindspore.ops.**mean**(*x*, *axis=None*, *keep_dims=False*)

Compute the mean(s) of the tensor along the specified axis(axes).

Parameters

- **x** (Tensor[*Number*]) –The input tensor.
- **axis** (*Union[int, tuple(int), list(int), Tensor]*) –Specify the axis(axes) for computation. If None , compute all elements in the *input* .
- **keep_dims** (*bool*) –Whether the output tensor has dim retained. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the mean of all elements.
>>> mindspore.ops.mean(input)
Tensor(shape=[], dtype=Int64, value= 4)
>>>
>>> # case 2: Compute the mean along axis 1.
>>> mindspore.ops.mean(input, axis=1)
Tensor(shape=[3], dtype=Int64, value= [5, 4, 4])
>>>
>>> # case 3: If keep_dims=True, the output shape will be same of that of the input.
>>> mindspore.ops.mean(input, axis=1, keep_dims=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[5],
 [4],
 [4]])
```

mindspore.ops.median

mindspore.ops.**median**(*input*, *axis=- 1*, *keepdims=False*)

Return the median(s) and indice(s) of the tensor along the specified axis.

Warning:

- *indices* does not necessarily contain the first occurrence of each median value found in the *input*, unless it is unique. The specific implementation of this API is device-specific. The results may be different on CPU and GPU.

Parameters

- **input** (Tensor) –The input tensor.

- **axis** (*int, optional*) –Specify the axis for computation. Default `-1` .
- **keepdims** (*bool, optional*) –Whether the output tensor has dim retained. Default `False` .

Returns

Tuple(median, median_indices) of 2 tensors.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the median along axis -1.
>>> mindspore.ops.median(input)
(Tensor(shape=[3], dtype=Int64, value= [4, 4, 3]),
 Tensor(shape=[3], dtype=Int64, value= [2, 3, 2]))
>>>
>>> # case 2: Compute the median along axis 0.
>>> mindspore.ops.median(input, axis=0)
(Tensor(shape=[4], dtype=Int64, value= [8, 2, 4, 5]),
 Tensor(shape=[4], dtype=Int64, value= [2, 1, 0, 0]))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.median(input, axis=0, keepdims=True)
(Tensor(shape=[1, 4], dtype=Int64, value=
 [[8, 2, 4, 5]]),
 Tensor(shape=[1, 4], dtype=Int64, value=
 [[2, 1, 0, 0]]))
```

mindspore.ops.min

`mindspore.ops.`**`min`**(*input*, *axis=None*, *keepdims=False*, ***, *initial=None*, *where=None*)

Return the minimum values and their indices along the given axis of the tensor.

Parameters

- **input** (`Tensor`) –The input tensor.
- **axis** (*int*) –Specify the axis for computation. If `None` , compute all elements in the *input* . Default `None` .
- **keepdims** (*bool*) –Whether the output tensor has dim retained. Default `False` .

Keyword Arguments

- **initial** (*scalar, optional*) –Initial value for the minimum. Default `None` .
- **where** (*Tensor[bool], optional*) –Specifies the range over which to compute the maximum values. The shape of this tensor must be broadcastable to the shape of *input* . An *initial* value must be specified. Default `None` , indicating that all elements are to be computed.

Returns

Tuple(min, min_indices) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[2, 5, 1, 6],
...                           [3, -7, -2, 4],
...                           [8, -4, 1, -3]])
>>> # case 1: By default, compute the minimum of all elements.
>>> mindspore.ops.min(input)
(Tensor(shape=[], dtype=Int64, value= -7),
 Tensor(shape=[], dtype=Int64, value= 0))
>>>
>>> # case 2: Compute minimum along axis 1.
>>> mindspore.ops.min(input, axis=1)
(Tensor(shape=[3], dtype=Int64, value= [ 1, -7, -4]),
 Tensor(shape=[3], dtype=Int64, value= [2, 1, 1]))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.min(input, axis=1, keepdims=True)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[ 1],
  [-7],
  [-4]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[2],
  [1],
  [1]]))
>>>
>>> # case 4: Use "where" to include only specific elements in computing the minimum.
>>> where = mindspore.tensor([[1, 0, 1, 0],
...                           [0, 0, 1, 1],
...                           [1, 1, 1, 0]], dtype=mindspore.bool_)
>>> mindspore.ops.min(input, axis=1, keepdims=True, initial=0, where=where)
(Tensor(shape=[3, 1], dtype=Int64, value=
 [[ 0],
  [-2],
  [-4]]),
 Tensor(shape=[3, 1], dtype=Int64, value=
 [[0],
  [2],
  [1]]))
>>>
>>> # case 5: The shape of "where" must be broadcast compatible with input.
>>> where = mindspore.tensor([[False],
...                           [False],
...                           [False]])
>>> mindspore.ops.min(input, axis=0, keepdims=True, initial=0, where=where)
(Tensor(shape=[1, 4], dtype=Int64, value=
 [[0, 0, 0, 0]]),
 Tensor(shape=[1, 4], dtype=Int64, value=
 [[0, 0, 0, 0]]))
```

mindspore.ops.norm

mindspore.ops.**norm**(*A*, *ord=None*, *dim=None*, *keepdim=False*, *, *dtype=None*)

Compute the matrix norm or vector norm of the tensor along a specified dimension.

ord is the calculation mode of norm. The following norm modes are supported.

ord	norm for matrices	norm for vectors
None (default)	Frobenius norm	*2*-norm (see below)
'fro'	Frobenius norm	–not supported –
'nuc'	nuclear norm	–not supported –
inf	$max(sum(abs(x), dim = 1))$	$max(abs(x))$
-inf	$min(sum(abs(x), dim = 1))$	$min(abs(x))$
0	–not supported –	$sum(x! = 0)$
1	$max(sum(abs(x), dim = 0))$	as below
-1	$min(sum(abs(x), dim = 0))$	as below
2	largest singular value	as below
-2	smallest singular value	as below
other *int* or *float*	–not supported –	$sum(abs(x)^{ord})^{(1/ord)}$

Parameters

- **A** (Tensor) –The input tensor.
- **ord** (*Union[int, float, inf, -inf, 'fro', 'nuc'], optional*) –Specify the kind of norm to take. Default None .
- **dim** (*Union[int, Tuple(int)], optional*) –Specify the dimension for computation. Default None .
  - If *dim* is int, calculate the vector norm.
  - if *dim* is a 2-tuple, calculate the matrix norm.
  - If *dim* is None and *ord* is None , flattened *A* to 1D and calculate 2-norm of the vector.
  - If *dim* is None and *ord* is not None, *A* must be 1D or 2D.
- **keepdim** (*bool*) –Whether the output tensor has dim retained. Default False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The data type returned. The data type returned. When set, *A* will be converted to the specified data type, before calculaing. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Note: Currently, complex numbers are not supported.

Examples

```
>>> import mindspore
>>> # Vector norms:
>>> A = mindspore.tensor([3., 4., 12.])
>>> mindspore.ops.norm(A)
Tensor(shape=[], dtype=Float32, value= 13)
>>> mindspore.ops.norm(A, ord=1)
Tensor(shape=[], dtype=Float32, value= 19)
>>> mindspore.ops.norm(A, ord=0)
Tensor(shape=[], dtype=Float32, value= 3)
>>>
>>> # Matrix norms:
>>> A = mindspore.tensor([[1., 2., 3.],
...                       [4., 5., 7.]])
>>> mindspore.ops.norm(A)  # Frobenius norm
Tensor(shape=[], dtype=Float32, value= 10.198)
>>> mindspore.ops.norm(A, ord='nuc')  # nuclear norm
Tensor(shape=[], dtype=Float32, value= 10.7625)
>>> mindspore.ops.norm(A, ord=1)  # 1-norm
Tensor(shape=[], dtype=Float32, value= 10)
>>>
>>> # Batched vector norm:
>>> mindspore.ops.norm(A, dim=1)
Tensor(shape=[2], dtype=Float32, value= [ 3.74165726e+00,  9.48683262e+00])
```

mindspore.ops.prod

mindspore.ops.**prod**(*input*, *axis=None*, *keep_dims=False*, *dtype=None*)

Return the product(s) of the tensor along the specified axis(axes).

Parameters

- **input** (*Tensor[Number]*) – The input tensor.
- **axis** (*Union[int, tuple(int), list(int), Tensor]*) – Specify the axis(axes) for computation. If None , compute all elements in the *input* . Default None .
- **keep_dims** (*bool*) – Whether the output tensor has dim retained. Default False .
- **dtype** (*mindspore.dtype*) – The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[9, 3, 4, 5],
...                           [5, 2, 7, 4],
...                           [8, 1, 3, 6]])
>>> # case 1: By default, compute the product of all elements.
>>> mindspore.ops.prod(input)
Tensor(shape=[], dtype=Int64, value= 21772800)
>>>
>>> # case 2: Compute the product along axis 1.
>>> mindspore.ops.prod(input, axis=1)
Tensor(shape=[3], dtype=Int64, value= [540, 280, 144])
>>>
>>> # case 3: If keep_dims=True, the output shape will be same of that of the input.
>>> mindspore.ops.prod(input, axis=1, keep_dims=True)
Tensor(shape=[3, 1], dtype=Int64, value=
[[540],
 [280],
 [144]])
```

mindspore.ops.std

mindspore.ops.**std**(*input*, *axis=None*, *ddof=0*, *keepdims=False*)

Compute the standard deviation of the tensor along a specified axis.

Parameters

- **input** (*Tensor[Number]*) – The input tensor.
- **axis** (*Union[int, tuple(int)], optional*) – Specify the axis for computation. If `None` , compute all elements in the *input* . Default `None` .
- **ddof** (*Union[int, bool], optional*) – Means Delta Degrees of Freedom. Default `0` .
  - If ddof is an integer, the divisor used in calculations is $N - ddof$, where N represents the number of elements.
  - If ddof is a boolean, `True` and `False` correspond to when ddof is an integer `1` and `0` respectively.
  - If ddof is 0, 1, True or False, the supported device is only Ascend and CPU. In other cases, the supported device is Ascend, GPU and CPU.
- **keepdims** (*bool, optional*) – Whether the output tensor has dim retained. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 3, 4, 2],
...                           [4, 2, 5, 3],
...                           [5, 4, 2, 3]])
>>> # case 1: By default, compute the standard deviation of all elements.
>>> output = mindspore.ops.std(input)
>>> print(output)
1.2133516
>>>
>>> # case 2: Compute the standard deviation along axis 0.
>>> output = mindspore.ops.std(input, axis=0)
>>> print(output)
[1.6996732 0.8164966 1.2472192 0.4714045]
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> output = mindspore.ops.std(input, axis=0, keepdims=True)
>>> print(output)
[[1.6996732 0.8164966 1.2472192 0.4714045]]
>>>
>>> # case 4: If ddof=1:
>>> output = mindspore.ops.std(input, axis=0, keepdims=True, ddof=1)
>>> print(output)
[[2.081666   1.         1.5275253  0.57735026]]
>>>
>>> # case 5: If ddof=True, same as ddof=1:
>>> output = mindspore.ops.std(input, axis=0, keepdims=True, ddof=True)
>>> print(output)
[[2.081666   1.         1.5275253  0.57735026]]
>>>
>>> # case 6: If ddof=False, same as ddof=0:
>>> output = mindspore.ops.std(input, axis=0, keepdims=True, ddof=False)
>>> print(output)
[[1.6996732 0.8164966 1.2472192 0.4714045]]
```

mindspore.ops.std_mean

mindspore.ops.**std_mean**(*input*, *axis=None*, *ddof=0*, *keepdims=False*)

Compute the standard deviation and the mean of the tensor along a specified axis.

Parameters

- **input** (*Tensor[Number]*) – The input tensor.
- **axis** (*Union[int, tuple(int)], optional*) – Specify the axis for computation. If `None`, compute all elements in the *input*. Default `None`.
- **ddof** (*Union[int, bool], optional*) – Means Delta Degrees of Freedom. Default `0`.
  - If ddof is an integer, the divisor used in calculations is $N - ddof$, where N represents the number of elements.
  - If ddof is a boolean, `True` and `False` correspond to when ddof is an integer `1` and `0` respectively.
  - If ddof is 0, 1, True or False, the supported device is only Ascend and CPU. In other cases, the supported device is Ascend, GPU and CPU.
- **keepdims** (*bool, optional*) – Whether the output tensor has dim retained. Default `False`.

Returns

Tuple(std, mean) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 3, 4, 2],
...                           [4, 2, 5, 3],
...                           [5, 4, 2, 3]])
>>> # case 1: By default, compute the standard deviation and mean of all elements.
>>> mindspore.ops.std_mean(input)
(Tensor(shape=[], dtype=Float32, value= 1.21335),
 Tensor(shape=[], dtype=Float32, value= 3.16667))
>>>
>>> # case 2: Compute the standard deviation and mean along axis 0.
>>> mindspore.ops.std_mean(input, axis=0)
(Tensor(shape=[4], dtype=Float32, value= [ 1.69967318e+00,  8.16496611e-01,  1.24721920e+00,␣
↪ 4.71404493e-01]),
 Tensor(shape=[4], dtype=Float32, value= [ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,␣
↪ 2.66666675e+00]))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> mindspore.ops.std_mean(input, axis=0, keepdims=True)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 1.69967318e+00,  8.16496611e-01,  1.24721920e+00,  4.71404493e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 4: If ddof=1:
>>> mindspore.ops.std_mean(input, axis=0, keepdims=True, ddof=1)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 2.08166599e+00,  1.00000000e+00,  1.52752531e+00,  5.77350259e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 5: If ddof=True, same as ddof=1:
>>> mindspore.ops.std_mean(input, axis=0, keepdims=True, ddof=True)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 2.08166599e+00,  1.00000000e+00,  1.52752531e+00,  5.77350259e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 6: If ddof=False, same as ddof=0:
>>> mindspore.ops.std_mean(input, axis=0, keepdims=True, ddof=False)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 1.69967318e+00,  8.16496611e-01,  1.24721920e+00,  4.71404493e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
```

mindspore.ops.var

mindspore.ops.**var**(*input*, *axis=None*, *ddof=0*, *keepdims=False*)

Compute the variance of the tensor along a specified axis.

Parameters

- **input** (*Tensor[Number]*) – The input tensor.
- **axis** (*Union[int, tuple(int)], optional*) – Specify the axis for computation. If `None` , compute all elements in the *input* . Default `None` .
- **ddof** (*Union[int, bool], optional*) – Means Delta Degrees of Freedom. Default `0` .
  - If ddof is an integer, the divisor used in calculations is $N - ddof$, where N represents the number of elements.
  - If ddof is a boolean, `True` and `False` correspond to when ddof is an integer `1` and `0` respectively.
  - If ddof is 0, 1, True or False, the supported device is only Ascend and CPU. In other cases, the supported device is Ascend, GPU and CPU.
- **keepdims** (*bool, optional*) – Whether the output tensor has dim retained. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 3, 4, 2],
...                           [4, 2, 5, 3],
...                           [5, 4, 2, 3]])
>>> # case 1: By default, compute the variance of all elements.
>>> output = mindspore.ops.var(input)
>>> print(output)
1.4722221
>>>
>>> # case 2: Compute the variance along axis 0.
>>> output = mindspore.ops.var(input, axis=0)
>>> print(output)
[2.8888888 0.6666667 1.5555557 0.2222222]
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
>>> output = mindspore.ops.var(input, axis=0, keepdims=True)
>>> print(output)
[[2.8888888 0.6666667 1.5555557 0.2222222]]
>>>
>>> # case 4: If ddof=1:
>>> output = mindspore.ops.var(input, axis=0, keepdims=True, ddof=1)
>>> print(output)
[[4.3333335 1.        2.3333335 0.3333333]]
>>>
>>> # case 5: If ddof=True, same as ddof=1:
>>> output = mindspore.ops.var(input, axis=0, keepdims=True, ddof=True)
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[4.3333335 1.        2.3333335 0.3333333]]
>>>
>>> # case 6: If ddof=False, same as ddof=0:
>>> output = mindspore.ops.var(input, axis=0, keepdims=True, ddof=False)
>>> print(output)
[[2.8888888 0.6666667 1.5555557 0.2222222]]
```

mindspore.ops.var_mean

mindspore.ops.**var_mean**(*input*, *axis=None*, *ddof=0*, *keepdims=False*)

Compute the variance and the mean of the tensor along a specified axis.

Parameters

- **input** (*Tensor[Number]*) – The input tensor.
- **axis** (*Union[int, tuple(int)], optional*) – Specify the axis for computation. If `None` , compute all elements in the *input* . Default `None` .
- **ddof** (*Union[int, bool], optional*) – Means Delta Degrees of Freedom. Default `0` .
  - If ddof is an integer, the divisor used in calculations is $N - ddof$, where N represents the number of elements.
  - If ddof is a boolean, `True` and `False` correspond to when ddof is an integer `1` and `0` respectively.
  - If ddof is 0, 1, True or False, the supported device is only Ascend and CPU. In other cases, the supported device is Ascend, GPU and CPU.
- **keepdims** (*bool, optional*) – Whether the output tensor has dim retained. Default `False` .

Returns

Tuple(var, mean) of 2 tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 3, 4, 2],
...                           [4, 2, 5, 3],
...                           [5, 4, 2, 3]])
>>> # case 1: By default, compute the variance and mean of all elements.
>>> mindspore.ops.var_mean(input)
(Tensor(shape=[], dtype=Float32, value= 1.47222),
 Tensor(shape=[], dtype=Float32, value= 3.16667))
>>>
>>> # case 2: Compute the variance and mean along axis 0.
>>> output = mindspore.ops.var_mean(input, axis=0)
(Tensor(shape=[4], dtype=Float32, value= [ 2.88888884e+00,  6.66666687e-01,  1.55555570e+00,␣
↪ 2.22222194e-01]),
 Tensor(shape=[4], dtype=Float32, value= [ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,␣
↪ 2.66666675e+00]))
>>>
>>> # case 3: If keepdims=True, the output shape will be same of that of the input.
```

(continues on next page)

(continued from previous page)

```
>>> output = mindspore.ops.var_mean(input, axis=0, keepdims=True)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 2.88888884e+00,  6.66666687e-01,  1.55555570e+00,  2.22222194e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 4: If ddof=1:
>>> output = mindspore.ops.var_mean(input, axis=0, keepdims=True, ddof=1)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 4.33333349e+00,  1.00000000e+00,  2.33333349e+00,  3.33333313e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 5: If ddof=True, same as ddof=1:
>>> output = mindspore.ops.var_mean(input, axis=0, keepdims=True, ddof=True)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 4.33333349e+00,  1.00000000e+00,  2.33333349e+00,  3.33333313e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
>>>
>>> # case 6: If ddof=False, same as ddof=0:
>>> output = mindspore.ops.var_mean(input, axis=0, keepdims=True, ddof=False)
(Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 2.88888884e+00,  6.66666687e-01,  1.55555570e+00,  2.22222194e-01]]),
 Tensor(shape=[1, 4], dtype=Float32, value=
 [[ 3.33333325e+00,  3.00000000e+00,  3.66666675e+00,  2.66666675e+00]]))
```

2.2.3 Comparison Functions

API Name	Description	Supported Platforms
mindspore.ops.argsort	Return the indices that sort the tensor along the specified axis.	Ascend GPU CPU
mindspore.ops.approximate_equal	Return a boolean tensor where two tensors are element-wise equal within a tolerance.	Ascend GPU CPU
mindspore.ops.bucketize	Return the indices of the buckets to which each element in the input tensor belongs.	Ascend GPU CPU
mindspore.ops.eq	Compute the equivalence of the two inputs element-wise.	Ascend GPU CPU
mindspore.ops.equal	Compute the equivalence of the two inputs element-wise.	Ascend GPU CPU
mindspore.ops.ge	Compute the value of $input >= other$ element-wise.	Ascend GPU CPU
mindspore.ops.greater	Compute the value of $input > other$ element-wise.	Ascend GPU CPU
mindspore.ops.greater_equal	Compute the value of $input >= other$ element-wise.	Ascend GPU CPU
mindspore.ops.gt	Compute the value of $input > other$ element-wise.	Ascend GPU CPU
mindspore.ops.intopk	Return whether the elements in second input tensor exist among the top k elements of the first input tensor.	Ascend GPU CPU

continues on next page

Table 8 – continued from previous page

mindspore.ops.isclose	Return a boolean tensor where two tensors are element-wise equal within a tolerance.	Ascend GPU CPU
mindspore.ops.isfinite	Return a boolean tensor indicating which elements are finite.	Ascend GPU CPU
mindspore.ops.isinf	Return a boolean tensor indicating which elements are +/- inifnity.	Ascend CPU GPU
mindspore.ops.isnan	Return a boolean tensor indicating which elements are NaN.	Ascend GPU CPU
mindspore.ops.isneginf	Return a boolean tensor indicating which elements are negative infinity.	Ascend GPU CPU
mindspore.ops.isposinf	Return a boolean tensor indicating which elements are positive infinity.	Ascend GPU CPU
mindspore.ops.isreal	Return a boolean tensor indicating which elements are real.	GPU CPU
mindspore.ops.is_complex	Return a boolean tensor indicating which elements are complex.	Ascend GPU CPU
mindspore.ops.is_floating_point	If the data type of the tensor is a floating point data type, return True.	Ascend GPU CPU
mindspore.ops.le	Compute the value of *input* <= *other* element-wise.	Ascend GPU CPU
mindspore.ops.less	Compute the value of *input* < *other* element-wise.	Ascend GPU CPU
mindspore.ops.less_equal	Compute the value of *input* <= *other* element-wise.	Ascend GPU CPU
mindspore.ops.lt	Alias for mindspore.ops.less() .	Ascend GPU CPU
mindspore.ops.maximum	Compute the maximum of the two input tensors element-wise.	Ascend GPU CPU
mindspore.ops.minimum	Compute the minimum of the two input tensors element-wise.	Ascend GPU CPU
mindspore.ops.msort	Return a tensor obtained by sorting the input tensor in ascending order along its first dimension.	Ascend GPU CPU
mindspore.ops.ne	Compute the non-equivalence of two inputs element-wise.	Ascend GPU CPU
mindspore.ops.not_equal	Alias for mindspore.ops.ne() .	Ascend GPU CPU
mindspore.ops.searchsorted	Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.	Ascend GPU CPU
mindspore.ops.topk	Return the top *k* largest or smallest elements of the input tensor along a specified dimension.	Ascend GPU CPU

mindspore.ops.argsort

mindspore.ops.**argsort**(*input*, *axis=- 1*, *descending=False*)

Return the indices that sort the tensor along the specified axis.

Note: The Ascend backend only supports sorting the last dimension.

Parameters

- **input** (Tensor) – The input tensor.

- **axis** (*int*) −Specify the axis to sort along. Default `-1` .
- **descending** (*bool*) −Specify the sorting order (ascending or descending).

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: 1-dimensional sort
>>> input = mindspore.tensor([1, 3, 5, 4, 2, 1])
>>> mindspore.ops.argsort(input)
Tensor(shape=[6], dtype=Int32, value= [0, 5, 4, 1, 3, 2])
>>>
>>> # case 2: multi-dimensional sort
>>> input = mindspore.tensor([[2, 1, 3],
...                           [6, 4, 3]])
>>> mindspore.ops.argsort(input, axis=1)
Tensor(shape=[2, 3], dtype=Int32, value=
[[1, 0, 2],
 [2, 1, 0]])
>>> mindspore.ops.argsort(input, axis=1, descending=True)
Tensor(shape=[2, 3], dtype=Int32, value=
[[2, 0, 1],
 [0, 1, 2]])
```

mindspore.ops.approximate_equal

`mindspore.ops.`**`approximate_equal`**(*x*, *y*, *tolerance=1e-5*)

Return a boolean tensor where two tensors are element-wise equal within a tolerance.

Support implicit type conversion and type promotion.

Math function is defined as:

$$out_i = \begin{cases} \text{if } |x_i - y_i| < \text{tolerance}, & True \\ \text{if } |x_i - y_i| \geq \text{tolerance}, & False \end{cases}$$

Two infinite values and two NaN values are not considered equal.

Parameters

- **x** (`Tensor`) −The first input tensor.
- **y** (`Tensor`) −The second input tensor.
- **tolerance** (*float*) −The maximum deviation that two elements can be considered equal. Default `1e-5` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> mindspore.ops.approximate_equal(mindspore.tensor([1e6, 2e6, float("inf"), float("-inf"),␣
↪float("nan")]),
...                                 mindspore.tensor([1e6, 2e7, float("inf"), float("-inf"),␣
↪float("nan")]))
Tensor(shape=[6], dtype=Bool, value= [ True, False, False, False, False])
>>>
>>> mindspore.ops.approximate_equal(mindspore.tensor([1e6, 2e6, 3e6]),
...                                 mindspore.tensor([1.00001e6, 2.00002e6, 3.00009e6]),␣
↪tolerance=1e3)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
>>>
>>> # If `x` and `y` have different datatypes, the lower precision data type will be␣
↪converted to the
    relatively highest precision data type.
>>> mindspore.ops.approximate_equal(mindspore.tensor([1, 2], mindspore.int32),
...                                 mindspore.tensor([1., 2], mindspore.float32))
Tensor(shape=[2], dtype=Bool, value= [ True,  True])
```

mindspore.ops.bucketize

mindspore.ops.**bucketize**(*input*, *boundaries*, *, *right=False*)

Return the indices of the buckets to which each element in the input tensor belongs. If *right* is `False`, the left boundary is open. For each element x in *input*, the returned index satisfies the following rules:

$$\begin{cases} boundaries[i-1] < x <= boundaries[i], & \text{if right} = False \\ boundaries[i-1] <= x < boundaries[i], & \text{if right} = True \end{cases}$$

Parameters

- **input** (Tensor) –The input tensor.
- **boundaries** (*list*) –A sorted ascending list of bucket boundary values.

Keyword Arguments

right (*bool, optional*) –if `False`, gets the lower bound index for each value in input from boundaries; If `True`, gets the upper bound index instead. Default: `False`.

Returns

Tensor

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[3, 6, 9], [3, 6, 9]])
>>> boundaries = [1., 3., 5., 7., 9.]
>>> output = mindspore.ops.bucketize(input, boundaries, right=True)
>>> output
Tensor(shape=[2, 3], dtype=Int32, value=
[[2, 3, 5],
 [2, 3, 5]])
```

mindspore.ops.eq

mindspore.ops.**eq**(*input*, *other*)

Compute the equivalence of the two inputs element-wise.

$$out_i = \begin{cases} \text{True, if } input_i = other_i \\ \text{False, if } input_i \neq other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The input must be two Tensors, or a Tensor and a Scalar.
- The shapes of the inputs can be broadcasted to each other.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **other** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> x = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.eq(x, 2.0)
>>> print(output)
[False  True False]
>>> # case 2: The shape of two inputs are the same
>>> x = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> y = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.eq(x, y)
>>> print(output)
[ True  True False]
```

mindspore.ops.equal

mindspore.ops.**equal**(*input*, *other*)

Compute the equivalence of the two inputs element-wise.

$$out_i = \begin{cases} \text{True, if } input_i = other_i \\ \text{False, if } input_i \neq other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The input must be two Tensors, or a Tensor and a Scalar.
- The shapes of the inputs can be broadcasted to each other.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **other** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.equal(input, 2.0)
>>> print(output)
[False  True False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.equal(input, other)
>>> print(output)
[ True  True False]
```

mindspore.ops.ge

mindspore.ops.**ge**(*input*, *other*)

Compute the value of $input >= other$ element-wise.

Note:

- Support implicit type conversion.
- The inputs must be two tensors or one tensor and one scalar.

- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them can be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If the input Tensor can be broadcast, the low dimension will be extended to the corresponding high dimension in another input by copying the value of the dimension.

$$out_i = \begin{cases} \text{True, if } input_i >= other_i \\ \text{False, if } input_i < other_i \end{cases}$$

Parameters

- **input** (*Union[Tensor, Number, bool]*) –The first input.
- **other** (*Union[Tensor, Number, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.ge(input, 2.0)
>>> print(output)
[False  True True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.ge(input, other)
>>> print(output)
[ True  True False]
```

mindspore.ops.greater

mindspore.ops.**greater**(*input*, *other*)

Compute the value of $input > other$ element-wise.

Parameters

- **input** (*Union[Tensor, Number]*) –The first input.
- **other** (*Union[Tensor, Number]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.greater(input, 2.0)
>>> print(output)
[False False True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.greater(input, other)
>>> print(output)
[ False False False]
```

mindspore.ops.greater_equal

mindspore.ops.**greater_equal**(*input*, *other*)

Compute the value of $input >= other$ element-wise.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **other** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.greater_equal(input, 2.0)
>>> print(output)
[False  True True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.greater_equal(input, other)
>>> print(output)
[ True  True False]
```

mindspore.ops.gt

mindspore.ops.**gt**(*input*, *other*)

Compute the value of $input > other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i > other_i \\ \text{False, if } input_i <= other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The inputs must be two tensors or one tensor and one scalar.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them can be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If the input Tensor can be broadcast, the low dimension will be extended to the corresponding high dimension in another input by copying the value of the dimension.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – The first input.
- **other** (*Union[Tensor, number.Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.gt(input, 2.0)
>>> print(output)
[False False True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.gt(input, other)
>>> print(output)
[ False False False]
```

mindspore.ops.intopk

mindspore.ops.**intopk**(*x1*, *x2*, *k*)

Return whether the elements in second input tensor exist among the top *k* elements of the first input tensor.

Parameters

- **x1** (Tensor) –The 2-D input tensor.
- **x2** (Tensor) –The 1-D input tensor, should satisfy $x2.shape[0] = x1.shape[0]$.
- **k** (*int*) –Top *k* elements.

Returns

A 1-D tensor whose data type is bool, has the same shape with *x2*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x1 = mindspore.tensor([[1, 8, 5, 2, 7], [4, 9, 1, 3, 5]], mindspore.float32)
>>> x2 = mindspore.tensor([1, 3], mindspore.int32)
>>> mindspore.ops.intopk(x1, x2, 3)
Tensor(shape=[2], dtype=Bool, value= [ True, False])
```

mindspore.ops.isclose

mindspore.ops.**isclose**(*input*, *other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Return a boolean tensor where two tensors are element-wise equal within a tolerance. Math function is defined as:

$$|input - other| atol + rtol » |other|$$

Two Infinite values are considered equal if they have the same sign, Two NaN values are considered equal if *equal_nan* is True .

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.
- **rtol** (*Union[float, int, bool], optional*) –Relative tolerance. Default 1e-05 .
- **atol** (*Union[float, int, bool], optional*) –Absolute tolerance. Default 1e-08 .
- **equal_nan** (*bool, optional*) –Whether two NaNs are considered equal. Default False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.isclose(mindspore.tensor([2e6, float("inf"), float("-inf"), float("inf"),␣
↪float("nan")]),
...                       mindspore.tensor([2e7, float("inf"), float("-inf"), float("-inf"),␣
↪float("nan")]))
Tensor(shape=[5], dtype=Bool, value= [False,  True,  True, False, False])
>>>
>>> mindspore.ops.isclose(mindspore.tensor([1e6, 2e6, 3e6]),
...                       mindspore.tensor([1.00008e6, 2.00008e7, 3.00008e8]), rtol=1e3)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
>>>
>>> mindspore.ops.isclose(mindspore.tensor([1e6, 2e6, 3e6]),
...                       mindspore.tensor([1.00001e6, 2.00002e6, 3.00009e6]), atol=1e3)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
>>> mindspore.ops.isclose(mindspore.tensor([float("nan"), 1, 2]),
...                       mindspore.tensor([float("nan"), 1, 2]), equal_nan=True)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
```

mindspore.ops.isfinite

mindspore.ops.**isfinite**(*input*)

Return a boolean tensor indicating which elements are finite.

An element is considered finite if it is not `NaN` , `-INF` , or `INF` .

Parameters

input (`Tensor`) −The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.ops.isfinite(input)
Tensor(shape=[5], dtype=Bool, value= [ True,  True, False, False, False])
```

mindspore.ops.isinf

mindspore.ops.**isinf**(*input*)

Return a boolean tensor indicating which elements are +/- inifnity.

> **Warning:**
>
> - This is an experimental API that is subject to change.
> - For Ascend, it is only supported on platforms above Atlas A2.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend CPU GPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.ops.isinf(input)
Tensor(shape=[5], dtype=Bool, value= [False, False,  True,  True, False])
```

mindspore.ops.isnan

mindspore.ops.**isnan**(*input*)

Return a boolean tensor indicating which elements are NaN.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.ops.isnan(input)
Tensor(shape=[5], dtype=Bool, value= [False, False, False, False,  True])
```

mindspore.ops.isneginf

mindspore.ops.**isneginf**(*input*)

Return a boolean tensor indicating which elements are negative infinity.

> **Warning:** For Ascend, it is only supported on platforms above Atlas A2.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.ops.isneginf(input)
Tensor(shape=[5], dtype=Bool, value= [False, False, False,  True, False])
```

mindspore.ops.isposinf

mindspore.ops.**isposinf**(*input*)

Return a boolean tensor indicating which elements are positive infinity.

> **Warning:** For Ascend, it is only supported on platforms above Atlas A2.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.ops.isposinf(input)
Tensor(shape=[5], dtype=Bool, value= [False, False,  True, False, False])
```

mindspore.ops.isreal

mindspore.ops.**isreal**(*input*)

Return a boolean tensor indicating which elements are real.

A complex value is considered real when its imaginary part is 0.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([False, 0j, 1, 2.1, 1+2j])
>>> mindspore.ops.isreal(input)
Tensor(shape=[5], dtype=Bool, value= [ True,  True,  True,  True, False])
```

mindspore.ops.is_complex

mindspore.ops.**is_complex**(*input*)

Return a boolean tensor indicating which elements are complex.

Parameters

input (Tensor) – The input tensor.

Returns

Bool

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 1+1j, 2+2j], mindspore.dtype.complex64)
>>> mindspore.ops.is_complex(input)
True
>>>
>>> input = mindspore.tensor([1, 1+1j, 2+2j], mindspore.dtype.complex128)
>>> mindspore.ops.is_complex(input)
True
>>> input = mindspore.tensor([1, 1+1j, 2+2j], mindspore.dtype.int32)
>>> mindspore.ops.is_complex(input)
False
```

mindspore.ops.is_floating_point

mindspore.ops.**is_floating_point**(*input*)

If the data type of the tensor is a floating point data type, return True. Otherwise return False.

Parameters

input (Tensor) −The input Tensor.

Returns

Bool

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([False, 0j, 1, 2.1, 1+2j], mindspore.float64)
>>> mindspore.ops.is_floating_point(input)
True
>>>
>>> input = mindspore.tensor([False, 0j, 1, 2.1, 1+2j], mindspore.float32)
>>> mindspore.ops.is_floating_point(input)
True
>>>
>>> input = mindspore.tensor([False, 0j, 1, 2.1, 1+2j], mindspore.float16)
>>> mindspore.ops.is_floating_point(input)
True
>>>
>>> input = mindspore.tensor([False, 0j, 1, 2.1, 1+2j], mindspore.int32)
>>> mindspore.ops.is_floating_point(input)
False
```

mindspore.ops.le

mindspore.ops.**le**(*input*, *other*)

Compute the value of $input <= other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i <= other_i \\ \text{False, if } input_i > other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The inputs must be two tensors or one tensor and one scalar.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.le(input, 2.0)
>>> print(output)
[True  True False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.le(input, other)
>>> print(output)
[ True  True  True]
```

mindspore.ops.less

mindspore.ops.**less**(*input*, *other*)

Compute the value of $input < other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i < other_i \\ \text{False, if } input_i >= other_i \end{cases}$$

Note: Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.less(input, 2.0)
>>> print(output)
[True  False False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.less(input, other)
>>> print(output)
[ False  False  True]
```

mindspore.ops.less_equal

`mindspore.ops.`**`less_equal`**(*input*, *other*)

Compute the value of $input <= other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i <= other_i \\ \text{False, if } input_i > other_i \end{cases}$$

Note:

- Support implicit type conversion.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.less_equal(input, 2.0)
>>> print(output)
[True  True False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.less_equal(input, other)
>>> print(output)
[ True  True  True]
```

mindspore.ops.lt

mindspore.ops.**lt**(*input*, *other*)

Alias for *mindspore.ops.less()*.

Supported Platforms:

Ascend GPU CPU

mindspore.ops.maximum

mindspore.ops.**maximum**(*input*, *other*)

Compute the maximum of the two input tensors element-wise.

$$output_i = \max(input_i, other_i)$$

Note:

- Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If one of the elements being compared is a NaN, then that element is returned.

Warning: If all inputs are scalar of integers. In Graph mode, the output will be Tensor of int32, while in PyNative mode, the output will be Tensor of int64.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1 : same data type
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.float32)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float32)
>>> mindspore.ops.maximum(input, other)
Tensor(shape=[3], dtype=Float32, value= [ 4.00000000e+00,  5.00000000e+00,  6.00000000e+00])
>>>
>>> # case 2 : the data type is the one with higher precision or higher digits among the two␣
↪inputs.
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.int64)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float64)
>>> mindspore.ops.maximum(input, other)
Tensor(shape=[3], dtype=Float64, value= [ 4.00000000e+00,  5.00000000e+00,  6.00000000e+00])
```

mindspore.ops.minimum

`mindspore.ops.`**`minimum`**(*input*, *other*)

Compute the minimum of the two input tensors element-wise.

$$output_i = \min(input_i, other_i)$$

Note:

- Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Shapes of them are supposed to be broadcast.
- If one of the elements being compared is a NaN, then that element is returned.

Parameters

- **`input`** (*Union[Tensor, Number, bool]*) – The first input.
- **`other`** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1 : same data type
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.float32)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float32)
>>> mindspore.ops.minimum(input, other)
Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00])
>>>
>>> # case 2 : the data type is the one with higher precision or higher digits among the two␣
 ↪inputs.
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.int64)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float64)
>>> mindspore.ops.minimum(input, other)
Tensor(shape=[3], dtype=Float64, value= [ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00])
```

mindspore.ops.msort

mindspore.ops.**msort**(*input*)

Return a tensor obtained by sorting the input tensor in ascending order along its first dimension.

ops.msort(input) is equivalent to *ops.sort(axis=0)(input)[0]*. See also `mindspore.ops.Sort()` for more details.

Parameters

input (Tensor) – The input tensor to sort.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[8, 2, 1],
...                           [5, 9, 3],
...                           [4, 6, 7]])
>>> mindspore.ops.msort(input)
Tensor(shape=[3, 3], dtype=Int64, value=
[[4, 2, 1],
 [5, 6, 3],
 [8, 9, 7]])
>>> # is equivalent to `ops.sort(axis=0)(input)[0]`
>>> mindspore.ops.sort(input, axis=0)[0]
Tensor(shape=[3, 3], dtype=Int64, value=
[[4, 2, 1],
 [5, 6, 3],
 [8, 9, 7]])
```

mindspore.ops.ne

mindspore.ops.**ne**(*input*, *other*)

Compute the non-equivalence of two inputs element-wise.

$$out_i = \begin{cases} \text{True, if } input_i \neq other_i \\ \text{False, if } input_i = other_i \end{cases}$$

Note:

- Support implicit type conversion.
- When the inputs are two tensors, the shapes of them could be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.ops.ne(input, 2.0)
>>> print(output)
[True False  True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.ops.ne(input, other)
>>> print(output)
[ False False  True]
```

mindspore.ops.not_equal

mindspore.ops.**not_equal**(*input*, *other*)

Alias for *mindspore.ops.ne()* .

Supported Platforms:

Ascend GPU CPU

mindspore.ops.searchsorted

mindspore.ops.**searchsorted**(*sorted_sequence*, *values*, *, *out_int32=False*, *right=False*, *side=None*, *sorter=None*)

Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.

Parameters

- **sorted_sequence** (Tensor) –The input tensor. If *sorter* not provided, it must contain a increasing sequence on the innermost dimension.
- **values** (Tensor) –The value that need to be inserted.

Keyword Arguments

- **out_int32** (bool, *optional*) –Whether the output datatype will be mindspore.int32. if False , the output datatype will be mindspore.int64. Default False .
- **right** (bool, *optional*) –Search Strategy. If True , return the last suitable index found; if False , return the first such index. Default False .
- **side** (*str*, *optional*) –the same as right but preferred. left corresponds to False for *right* and right corresponds to True for *right*. An error will be reported if this parameter is set to left while *right* is True. Default None .
- **sorter** (Tensor, *optional*) –An index sequence sorted in ascending order along the innermost dimension of *sorted_sequence* , which is used together with the unsorted *sorted_sequence* . CPU and GPU only support None. Default None .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> sorted_sequence = mindspore.tensor([1, 2, 2, 3, 4, 5, 5])
>>> values = mindspore.tensor([2])
>>> mindspore.ops.searchsorted(sorted_sequence, values)
Tensor(shape=[1], dtype=Int64, value= [1])
```

mindspore.ops.topk

mindspore.ops.**topk**(*input*, *k*, *dim=None*, *largest=True*, *sorted=True*)

Return the top *k* largest or smallest elements of the input tensor along a specified dimension.

Warning:

- If sorted is set to False, it will use the aicpu operator, the performance may be reduced. In addition, due to different memory layout and traversal methods on different platforms, the display order of calculation results may be inconsistent when *sorted* is False.

Parameters

- **input** (Tensor) –The input tensor.
- **k** (*int*) –The number elements to be returned.
- **dim** (*int, optional*) –Specify the dimension for sorting. Default None .
- **largest** (*bool, optional*) –If True , return largest elements. If False , then return smallest elements. Default True .
- **sorted** (*bool, optional*) –If True , the elements are returned in descending order. If False , the obtained elements will not be sorted. Default True .

Returns

Tuple(values, indices) of 2 tensors.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[8, 2, 1],
...                           [5, 9, 3],
...                           [4, 6, 7]])
>>> # case 1: If dim is not given, the last dimension of the input is chosen.
>>> mindspore.ops.topk(input, 2)
(Tensor(shape=[3, 2], dtype=Int64, value=
 [[8, 2],
  [9, 5],
  [7, 6]]),
 Tensor(shape=[3, 2], dtype=Int32, value=
 [[0, 1],
  [1, 0],
  [2, 1]]))
>>> # case 2: when dim is 0:
>>> mindspore.ops.topk(input, 2, dim=0)
(Tensor(shape=[2, 3], dtype=Int64, value=
[[8, 9, 7],
 [5, 6, 3]]),
Tensor(shape=[2, 3], dtype=Int32, value=
[[0, 1, 2],
 [1, 2, 1]]))
```

(continues on next page)

(continued from previous page)

```
>>> # case 3: when largest is False, return smallest values.
>>> mindspore.ops.topk(input, 2, dim=0, largest=False)
(Tensor(shape=[2, 3], dtype=Int64, value=
 [[4, 2, 1],
  [5, 6, 3]]),
 Tensor(shape=[2, 3], dtype=Int32, value=
 [[2, 0, 0],
  [1, 2, 1]]))
```

2.2.4 Linear Algebraic Functions

API Name	Description	Supported Platforms
mindspore.ops.addbmm	Apply batch matrix multiplication to *batch1* and *batch2*, with a reduced add step and add *input* to the result.	Ascend GPU CPU
mindspore.ops.addmm	Multiply matrix *mat1* and matrix *mat2*.	Ascend GPU CPU
mindspore.ops.addr	Compute the outer product of two vector *vec1* and *vec2*, and add the resulting matrix to *x*.	Ascend GPU CPU
mindspore.ops.adjoint	Calculate the conjugation of tensor element-wise, and transpose the last two dimensions.	Ascend GPU CPU
mindspore.ops.baddbmm	Perform a batch matrix-matrix product of matrices in *batch1* and *batch2* , *input* is added to the final result.	Ascend GPU CPU
mindspore.ops.batch_dot	Computation of batch dot product between samples in two tensors containing batch dims.	Ascend GPU CPU
mindspore.ops.bmm	Perform a batch matrix-matrix product of matrices in input tensors.	Ascend GPU CPU
mindspore.ops.cholesky	Return the Cholesky decomposition of zero or more batch dimensions consisting of symmetric positive-definite matrices.	GPU CPU
mindspore.ops.cholesky_solve	Computes the solution of a set of linear equations with a positive definite matrix, according to its Cholesky decomposition factor *input2* .	Ascend GPU CPU
mindspore.ops.cond	Return the matrix norm or vector norm of a given tensor.	GPU CPU
mindspore.ops.dot	Computation a dot product of two input tensors.	Ascend GPU CPU
mindspore.ops.eigvals	Compute the eigenvalues of a square matrix.	Ascend CPU
mindspore.ops.geqrf	Perform a QR decomposition on the input tensor $A = QR$.	Ascend GPU CPU
mindspore.ops.ger	Calculate the outer product of two arrays *input* and *vec2*.	Ascend GPU CPU
mindspore.ops.inner	Return the dot product of two 1-D tensors.	Ascend GPU CPU
mindspore.ops.inverse	Compute the inverse of the input matrix.	GPU CPU
mindspore.ops.kron	Compute the Kronecker product of two tensors.	Ascend GPU CPU
mindspore.ops.logdet	Calculate log determinant of one or a batch of square matrices.	CPU
mindspore.ops.lu_solve	Compute the solution to a system of linear equations $Ay = b$.	Ascend GPU CPU

continues on next page

Table 9 – continued from previous page

mindspore.ops.lu_unpack	Unpack the LU decomposition returned by mindspore.scipy.linalg.lu_factor() into the P, L, U matrices.	GPU CPU
mindspore.ops.matmul	Return the matrix product of two tensors.	Ascend GPU CPU
mindspore.ops.matrix_band_part	Copy a tensor setting everything outside a central band in each innermost matrix to zero.	Ascend GPU CPU
mindspore.ops.matrix_solve	Solves systems of linear equations.	Ascend CPU
mindspore.ops.matrix_diag	Return a tensor with the contents in *x* as k[0]-th to k[1]-th diagonals of a matrix, with everything else padded with *padding_value* .	Ascend GPU CPU
mindspore.ops.matrix_diag_part	Return a tensor that retains the values of the specified diagonal while setting all other elements to zero.	Ascend GPU CPU
mindspore.ops.matrix_set_diag	Return a tensor by replacing the elements on the k[0]-th to k[1]-th diagonals of the matrix *x* with the values from the input *diagonal* .	Ascend GPU CPU
mindspore.ops.mm	Returns the matrix product of two arrays.	Ascend GPU CPU
mindspore.ops.mv	Multiplies matrix *mat* and vector *vec*.	Ascend GPU CPU
mindspore.ops.outer	Compute outer product of two tensors.	Ascend GPU CPU
mindspore.ops.orgqr	Compute the first *N* columns of a product of Householder matrices.	Ascend GPU CPU
mindspore.ops.ormqr	Calculates the product of a matrix *other* and a matrix Q (represented by Householder vectors *input* and Householder reflection coefficients *tau*).	GPU
mindspore.ops.pinv	Computes the (Moore-Penrose) pseudo-inverse of a matrix.	CPU
mindspore.ops.svd	Computes the singular value decompositions of one or more matrices.	Ascend GPU CPU
mindspore.ops.slogdet	Computes the sign and the log of the absolute value of the determinant of one or more square matrices.	Ascend GPU CPU
mindspore.ops.trace	Return the sum of the elements along the diagonal of the input tensor.	Ascend GPU CPU
mindspore.ops.tensor_dot	Compute the tensor dot product along the specified axes.	Ascend GPU CPU
mindspore.ops.vander	Generates a Vandermonde matrix.	Ascend GPU CPU
mindspore.ops.vecdot	Calculates the dot product of two batches of vectors along the specified dimension.	Ascend GPU CPU

mindspore.ops.addbmm

mindspore.ops.**addbmm**(*input*, *batch1*, *batch2*, *, *beta=1*, *alpha=1*)

Apply batch matrix multiplication to *batch1* and *batch2*, with a reduced add step and add *input* to the result.

Note:

- *batch1* and *batch2* must be 3-D tensors each containing the same number of matrices.
- When batch1 is a (C, W, T) tensor and batch2 is a (C, T, H) tensor, input must be broadcastable with (W, H) tensor, and out will be a (W, H) tensor.
- If *beta* is 0, then *input* will be ignored.

$$output = \beta input + \alpha(\sum_{i=0}^{b-1} batch1_i @ batch2_i)$$

Parameters

- **input** (Tensor) –The input tensor.
- **batch1** (Tensor) –The first batch of tensor to be multiplied.
- **batch2** (Tensor) –The second batch of tensor to be multiplied.

Keyword Arguments

- **beta** (*Union[int, float], optional*) –Scale factor for *input*. Default 1 .
- **alpha** (*Union[int, float], optional*) –Scale factor for (*batch1* @ *batch2*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> m = mindspore.ops.ones((3, 3))
>>> arr1 = mindspore.tensor([[8., 7., 6.], [5., 4., 3.], [2., 1., 0.]])
>>> arr2 = mindspore.tensor([[5., 4., 3.], [2., 1., 0.], [8., 7., 6.]])
>>> output = mindspore.ops.addbmm(m, arr1, arr2)
>>> print(output)
[[172. 136. 100.]
 [172. 136. 100.]
 [172. 136. 100.]]
```

mindspore.ops.addmm

mindspore.ops.**addmm**(*input*, *mat1*, *mat2*, *, *beta=1*, *alpha=1*)

Multiply matrix *mat1* and matrix *mat2*. The matrix *input* is added to the final result.

Note:

- If *beta* is 0, then *input* will be ignored.

$$output = \beta input + \alpha(mat1 @ mat2)$$

Parameters

- **input** (Tensor) –The input tensor.
- **mat1** (Tensor) –The first tensor to be multiplied.
- **mat2** (Tensor) –The second tensor to be multiplied.

Keyword Arguments

- **beta** (*Union[int, float], optional*) –Scale factor for *input*. Default 1 .

- **alpha** (*Union[int, float], optional*) –Scale factor for (*mat1* @ *mat2*) . Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> m = mindspore.ops.ones((3, 3))
>>> arr1 = mindspore.tensor([[8., 7., 6.], [5., 4., 3.], [2., 1., 0.]])
>>> arr2 = mindspore.tensor([[5., 4., 3.], [2., 1., 0.], [8., 7., 6.]])
>>> output = mindspore.ops.addmm(m, arr1, arr2)
>>> print(output)
[[103.  82.  61.]
 [ 58.  46.  34.]
 [ 13.  10.   7.]]
```

mindspore.ops.addr

mindspore.ops.**addr**(*x*, *vec1*, *vec2*, *, *beta=1*, *alpha=1*)

Compute the outer product of two vector *vec1* and *vec2*, and add the resulting matrix to *x*.

Note:

- Given *vec1* and *vec2* of sizes N and M, *x* must be able to broadcast to a matrix of shape (N, M), and out will be a matrix of shape (N, M) .
- Setting *beta* to 0 will exclude *x* from the computation.

$$output = x + (vec1vec2)$$

Parameters

- **x** (Tensor) –Vector to be added.
- **vec1** (Tensor) –The first tensor to be multiplied.
- **vec2** (Tensor) –The second tensor to be multiplied.

Keyword Arguments

- **beta** (*scalar[int, float, bool], optional*) –Scale factor for *x* Default 1 .
- **alpha** (*scalar[int, float, bool], optional*) –Scale factor for (*vec1* ⊗ *vec2*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[2., 2.], [3., 2.], [3., 4.]])
>>> vec1 = mindspore.tensor([2., 3., 2.])
>>> vec2 = mindspore.tensor([3., 4.])
>>> output = mindspore.ops.addr(x, vec1, vec2)
>>> print(output)
[[ 8. 10.]
 [12. 14.]
 [ 9. 12.]]
```

mindspore.ops.adjoint

mindspore.ops.**adjoint**(*x*)

Calculate the conjugation of tensor element-wise, and transpose the last two dimensions.

Parameters

x (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor(([[0. + 0.j, 1. + 1.j], [2. + 2.j, 3. + 3.j]]), mindspore.
↪complex128)
>>> mindspore.ops.adjoint(x)
Tensor(shape=[2, 2], dtype=Complex128, value=
[[0-0j, 2-2j],
 [1-1j, 3-3j]])
```

mindspore.ops.baddbmm

mindspore.ops.**baddbmm**(*input*, *batch1*, *batch2*, *beta=1*, *alpha=1*)

Perform a batch matrix-matrix product of matrices in *batch1* and *batch2* , *input* is added to the final result.

Note:

- *batch1* and *batch2* must be 3-D tensors each containing the same number of matrices.
- When batch1 is a (C, W, T) tensor and batch2 is a (C, T, H) tensor, input must be broadcastable with (C, W, H) tensor, and out will be a (C, W, H) tensor.
- If *beta* is 0, then *input* will be ignored.
- *beta* and *alpha* must be integers when inputs of type not FloatTensor.

$$\text{out}_i = \beta\text{input}_i + \alpha(\text{batch1}_i \text{ @ } \text{batch2}_i)$$

Parameters

- **input** (Tensor) –The input tensor.
- **batch1** (Tensor) –The first batch of matrices to be multiplied.
- **batch2** (Tensor) –The second batch of matrices to be multiplied.
- **beta** (*Union[float, int], optional*) –Scale factor for *input*. Default 1 .
- **alpha** (*Union[float, int], optional*) –Scale factor for (*batch1* @ *batch2*). Default 1 .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.ones((3, 3))
>>> batch1 = mindspore.tensor([[8., 7., 6.], [5., 4., 3.], [2., 1., 0.]])
>>> batch2 = mindspore.tensor([[5., 4., 3.], [2., 1., 0.], [8., 7., 6.]])
>>> output = mindspore.ops.baddbmm(input, batch1, batch2)
>>> print(output)
[[103.  82.  61.]
 [ 58.  46.  34.]
 [ 13.  10.   7.]]
```

mindspore.ops.batch_dot

mindspore.ops.**batch_dot**(*x1*, *x2*, *axes=None*)

Computation of batch dot product between samples in two tensors containing batch dims.

Note:

- *x1* or *x2* first dimension is batch size. Datatype must be float32 and the rank must be greater than or equal to 2.

$$output = x1[batch, :] \text{ů} x2[batch, :]$$

Parameters

- **x1** (Tensor) –The first input tensor.
- **x2** (Tensor) –The second input tensor.
- **axes** (*Union[int, tuple(int), list(int)]*) –Specify the axes for computation. Default None .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> # case 1: axes is a tuple(axes of `x1` , axes of `x2` )
>>> x1 = mindspore.ops.ones([2, 2, 3])
>>> x2 = mindspore.ops.ones([2, 3, 2])
>>> axes = (-1, -2)
>>> output = mindspore.ops.batch_dot(x1, x2, axes)
>>> print(output)
[[[3. 3.]
  [3. 3.]]
 [[3. 3.]
  [3. 3.]]]
>>> print(output.shape)
(2, 2, 2)
>>> x1 = mindspore.ops.ones([2, 2], mindspore.float32)
>>> x2 = mindspore.ops.ones([2, 3, 2], mindspore.float32)
>>> axes = (1, 2)
>>> output = mindspore.ops.batch_dot(x1, x2, axes)
>>> print(output)
[[2. 2. 2.]
 [2. 2. 2.]]
>>> print(output.shape)
(2, 3)
>>>
>>> # case 2: axes is None
>>> x1 = mindspore.ops.ones([6, 2, 3, 4], mindspore.float32)
>>> x2 = mindspore.ops.ones([6, 5, 4, 8], mindspore.float32)
>>> output = mindspore.ops.batch_dot(x1, x2)
>>> print(output.shape)
(6, 2, 3, 5, 8)
>>>
>>> # case 3: axes is a int data.
>>> x1 = mindspore.ops.ones([2, 2, 4])
>>> x2 = mindspore.ops.ones([2, 5, 4, 5])
>>> output = mindspore.ops.batch_dot(x1, x2, 2)
>>> print(output.shape)
(2, 2, 5, 5)
```

mindspore.ops.bmm

`mindspore.ops.`**`bmm`**(*input_x*, *mat2*)

Perform a batch matrix-matrix product of matrices in input tensors.

$$\text{output}[..., :, :] = \text{matrix}(input_x[..., :, :]) * \text{matrix}(mat2[..., :, :])$$

The dim of *input_x* can not be less than *3* and the dim of *mat2* can not be less than *2*.

Parameters

- **`input_x`** (`Tensor`) −The first tensor to be multiplied.
- **`mat2`** (`Tensor`) −The second tensor to be multiplied.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.ops.arange(24, dtype=mindspore.float32).reshape(2, 4, 1, 3)
>>> mat2 = mindspore.ops.arange(72, dtype=mindspore.float32).reshape(2, 4, 3, 3)
>>> out = mindspore.ops.bmm(input_x, mat2)
>>> print(out)
[[[[  15.,   18.,   21.]],
  [[ 150.,  162.,  174.]],
  [[ 447.,  468.,  489.]],
  [[ 906.,  936.,  966.]]],
 [[[1527., 1566., 1605.]],
  [[2310., 2358., 2406.]],
  [[3255., 3312., 3369.]],
  [[4362., 4428., 4494.]]]]
```

mindspore.ops.cholesky

`mindspore.ops.`**`cholesky`**(*input_x*, *upper=False*)

Return the Cholesky decomposition of zero or more batch dimensions consisting of symmetric positive-definite matrices.

If *upper* is *True*, return an upper-triangular matrix, U, and the decomposition has the form:

$$A = U^T U$$

If *upper* is *False*, return a lower-triangular matrix, L, and the decomposition has the form:

$$A = LL^T$$

where A is the symmetric positive-definite matrix.

Parameters

- **`input_x`** (`Tensor`) −The input tensor of shape $(*, N, N)$, where $*$ is batch dimensions. In the above formula, A .
- **`upper`** (*`bool, optional`*) −Return an upper-triangular matrix or not. Default: `False` .

Returns

Tensor

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.tensor([[1.0, 1.0], [1.0, 2.0]])
>>> output = mindspore.ops.cholesky(input_x, upper=False)
>>> print(output)
[[1. 0.]
 [1. 1.]]
```

mindspore.ops.cholesky_solve

mindspore.ops.**cholesky_solve**(*input*, *input2*, *upper=False*)

Computes the solution of a set of linear equations with a positive definite matrix, according to its Cholesky decomposition factor *input2* .

If *upper* is set to `True` and *input2* is upper triangular, the output tensor is that:

$$output = (input2^T * input2)^{-1} input$$

If *upper* is set to `False` and *input2* is lower triangular, the output is that:

$$output = (input2 * input2^T)^{-1} input$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) –The input tensor of shape $(*, N, M)$.
- **input2** (`Tensor`) –The tensor of shape $(*, N, N)$, composed of upper or lower triangular Cholesky factor.
- **upper** (*bool, optional*) –Whether to treat the Cholesky factor as an upper triangular matrix. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input1 = mindspore.tensor([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])
>>> input2 = mindspore.tensor([[2., 0., 0.], [4., 1., 0.], [-1., 1., 2.]])
>>> out = mindspore.ops.cholesky_solve(input1, input2, upper=False)
>>> print(out)
[[ 5.8125 -2.625   0.625 ]
 [-2.625   1.25   -0.25  ]
 [ 0.625  -0.25    0.25  ]]
```

mindspore.ops.cond

mindspore.ops.**cond**(*A*, *p=None*)

Return the matrix norm or vector norm of a given tensor.

p is the calculation mode of norm. The following norm modes are supported.

p	norm for matrices	norm for vectors
None (default)	2-norm (see below)	2-norm (see below)
'fro'	Frobenius norm	–not supported –
'nuc'	nuclear norm	–not supported –
inf	$max(sum(abs(x), dim = 1))$	$max(abs(x))$
-inf	$min(sum(abs(x), dim = 1))$	$min(abs(x))$
0	–not supported –	$sum(x! = 0)$
1	$max(sum(abs(x), dim = 0))$	as below
-1	$min(sum(abs(x), dim = 0))$	as below
2	largest singular value	as below
-2	smallest singular value	as below
other int or float	–not supported –	$sum(abs(x)^p)^{(1/p)}$

Note: Currently, complex numbers are not supported.

Parameters

- **A** (Tensor) –The input tensor which is zero or more batch dimensions.
- **p** (*Union[int, float, inf, -inf, 'fro', 'nuc'], optional*) –Norm's mode. Refer to the table above for behavior. Default None.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1.0, 0.0, -1.0], [0.0, 1.0, 0.0], [1.0, 0.0, 1.0]])
>>> print(mindspore.ops.cond(x))
1.4142
>>> print(mindspore.ops.cond(x, 'fro'))
3.1622777
```

mindspore.ops.dot

mindspore.ops.**dot**(*input*, *other*)

Computation a dot product of two input tensors.

Note:

- Datatype of the input tensors must be float16 or float32, and the rank must be greater than or equal to 2.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.ops.ones([2, 3])
>>> other = mindspore.ops.ones([1, 3, 2])
>>> output = mindspore.ops.dot(input, other)
>>> print(output)
[[[3. 3.]]
 [[3. 3.]]]
>>> print(output.shape)
(2, 1, 2)
>>> input = mindspore.ops.ones([1, 2, 3])
>>> other = mindspore.ops.ones([1, 3, 2])
>>> output = mindspore.ops.dot(input, other)
>>> print(output)
[[[[3. 3.]]
  [[3. 3.]]]]
>>> print(output.shape)
(1, 2, 1, 2)
>>> input = mindspore.ops.ones([1, 2, 3])
>>> other = mindspore.ops.ones([2, 3, 2])
>>> output = mindspore.ops.dot(input, other)
>>> print(output)
[[[[3. 3.]
   [3. 3.]]
  [[3. 3.]
   [3. 3.]]]]
>>> print(output.shape)
(1, 2, 2, 2)
>>> input = mindspore.ops.ones([3, 2, 3])
>>> other = mindspore.ops.ones([2, 1, 3, 2])
>>> output = mindspore.ops.dot(input, other)
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[[[[3. 3.]]
   [[3. 3.]]]
  [[[3. 3.]]
   [[3. 3.]]]]
 [[[[3. 3.]]
   [[3. 3.]]]
  [[[3. 3.]]
   [[3. 3.]]]]
 [[[[3. 3.]]
   [[3. 3.]]]
  [[[3. 3.]]
   [[3. 3.]]]]]
>>> print(output.shape)
(3, 2, 2, 1, 2)
```

mindspore.ops.eigvals

mindspore.ops.**eigvals**(*A*)

Compute the eigenvalues of a square matrix.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

A (Tensor) –Square matrices with shape $(*, N, N)$.

Returns

Tensor

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> mindspore.ops.eigvals(mindspore.tensor([[1.0, 0.0], [0.0, 2.0]]))
Tensor(shape=[2], dtype=Complex64, value= [1+0j, 2+0j])
```

mindspore.ops.geqrf

mindspore.ops.**geqrf**(*input*)

Perform a QR decomposition on the input tensor $A = QR$.

The tensor is decomposed into the product of an orthogonal matrix Q and an upper triangular matrix R.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) – The input tensor.

Returns

Tuple(*y*, *tau*) of 2 tensors.

- **y** (Tensor) - Store the matrices *Q* and *R* implicitly. The matrix *Q* (Householder vectors) is stored below the diagonal, and the elements of matrix *R* are stored on and above the diagonal.
- **tau** (Tensor) - Store the scaling factors of each Householder transformation (Householder reflection coefficients).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-2.0, -1.0], [1.0, 2.0]])
>>> y, tau = mindspore.ops.geqrf(input)
>>> print(y)
[[ 2.236068   1.7888544]
 [-0.236068   1.3416407]]
>>> print(tau)
[1.8944271 0.       ]
```

mindspore.ops.ger

mindspore.ops.**ger**(*input*, *vec2*)

Calculate the outer product of two arrays *input* and *vec2*.

Note: Currently Ascend does not support float64 data input.

Parameters

- **input** (Tensor) – The 1-D input tensor.
- **vec2** (Tensor) – The 1-D input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1., 2., 3., 4.])
>>> vec2 = mindspore.tensor([1., 2., 3.])
>>> output = mindspore.ops.ger(input, vec2)
>>> print(output)
[[ 1.  2.  3.]
 [ 2.  4.  6.]
 [ 3.  6.  9.]
 [ 4.  8. 12.]]
```

mindspore.ops.inner

mindspore.ops.**inner**(*input*, *other*)

Return the dot product of two 1-D tensors.

For higher dimensions, return a sum product over the last axis.

Note: If *input* or *other* is a Tensor scalar, *mindspore.ops.inner()* will be the same as *mindspore.ops.mul()* .

Parameters

- **input** (Tensor) –The first input.
- **other** (Tensor) –The second input.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1: Two 1D tensors
>>> input = mindspore.tensor([1., 2., 3.])
>>> y = mindspore.tensor([4., 5., 6.])
>>> mindspore.ops.inner(input, y)
Tensor(shape=[], dtype=Float32, value= 32)
>>> # case2: Tensor scalar and tensor
>>> input = mindspore.tensor([[[1., 2., 3.], [3., 2., 1.]], [[4., 5., 6.], [4., 5., 6.]]])
>>> y = mindspore.tensor(2.)
>>> output = mindspore.ops.inner(input, y)
>>> print(output)
[[[ 2.  4.  6.]
  [ 6.  4.  2.]]
 [[ 8. 10. 12.]
  [ 8. 10. 12.]]]
>>> # case3: Two tensors
>>> input = mindspore.tensor([[[1., 2., 3.], [3., 2., 1.]], [[4., 5., 6.], [4., 5., 6.]]])
```

(continues on next page)

(continued from previous page)

```
>>> y = mindspore.tensor([[2., 3., 4.], [4., 3., 2.]])
>>> output = mindspore.ops.inner(input, y)
>>> print(output)
[[[20. 16.]
  [16. 20.]]
 [[47. 43.]
  [47. 43.]]]
```

mindspore.ops.inverse

mindspore.ops.**inverse**(*input*)

Compute the inverse of the input matrix.

Note: The *input* must be at least two dimensions, and the size of the last two dimensions must be the same size. The matrix must be invertible. Dtype of complex numbers is not supported.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> print(mindspore.ops.inverse(mindspore.tensor([[1., 2.], [3., 4.]])))
[[-2.   1. ]
 [ 1.5 -0.5]]
```

mindspore.ops.kron

mindspore.ops.**kron**(*input*, *other*)

Compute the Kronecker product of two tensors.

If the shape of *input* is (a_0 input a_1 input $\cdots$input a_n) and the shape of *other* is (b_0 input b_1 input $\cdots$input b_n) , the result will be ($a_0 * b_0$ input $a_1 * b_1$ input $\cdots$input $a_n * b_n$) .

$$(inputother)_{k_0,k_1,\ldots k_n} = input_{i_0,i_1,\ldots i_n} * other_{j_0,j_1,\ldots j_n},$$

where $k_t = i_t * b_t + j_t$ for $0 \leq t \leq n$.

Note: Supports real-valued and complex-valued inputs.

Parameters

- **input** (Tensor) –First input tensor.
- **other** (Tensor) –Second input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[0., 1., 2.], [3., 4., 5.]])
>>> other = mindspore.tensor([[-1., -2., -3.], [-4., -6., -8.]])
>>> output = mindspore.ops.kron(input, other)
>>> print(output)
[[  0.   0.   0.  -1.  -2.  -3.  -2.  -4.  -6.]
 [  0.   0.   0.  -4.  -6.  -8.  -8. -12. -16.]
 [ -3.  -6.  -9.  -4.  -8. -12.  -5. -10. -15.]
 [-12. -18. -24. -16. -24. -32. -20. -30. -40.]]
```

mindspore.ops.logdet

mindspore.ops.**logdet**(*input*)

Calculate log determinant of one or a batch of square matrices.

Parameters

input (Tensor) –The input tensor of shape $(*, N, N)$ where $*$ means batch dimensions.

Returns

Tensor

Supported Platforms:

CPU

Examples

```
>>> import mindspore
>>> a = mindspore.tensor([[[8., 9.], [1., 2.]], [[5., 6.], [3., 4.]]])
>>> mindspore.ops.logdet(a)
Tensor(shape=[2], dtype=Float32, value= [ 1.94591010e+00,  6.93146825e-01])
```

mindspore.ops.lu_solve

mindspore.ops.**lu_solve**(*b*, *LU_data*, *LU_pivots*)

Compute the solution to a system of linear equations $Ay = b$.

Note:

- *b* of shape $(*, m, k)$, *LU_data* of shape $(*, m, m)$, *LU_pivots* of shape $(*, m)$, where $*$ means batch dimensions.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **b** (Tensor) −Column vector *b* in the above equation.
- **LU_data** (Tensor) −LU decomposition. The *A* in the formula above.
- **LU_pivots** (Tensor) −Permutation matrix P of LU decomposition.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> b = mindspore.tensor([[1.], [3.], [3.]])
>>> LU_data = mindspore.tensor([[2., 1., 1.], [0.5, 1., 1.5], [0.5, 0., 2.5]])
>>> LU_pivots = mindspore.tensor(([2, 2, 3]), mindspore.int32)
>>> y = mindspore.ops.lu_solve(b, LU_data, LU_pivots)
>>> print(y)
[[ 1.9000001]
 [-1.4000001]
 [ 0.6      ]]
```

mindspore.ops.lu_unpack

mindspore.ops.**lu_unpack**(*LU_data*, *LU_pivots*, *unpack_data=True*, *unpack_pivots=True*)

Unpack the LU decomposition returned by *mindspore.scipy.linalg.lu_factor()* into the P, L, U matrices.

Note:

- *LU_data* of shape $(*, M, N)$, *LU_pivots* of shape $(*, min(M, N))$, where $*$ is batch dimensions.

Parameters

- **LU_data** (Tensor) −The packed LU factorization data, the rank is greater than or equal to 2.
- **LU_pivots** (Tensor) −The packed LU factorization pivots.

- **unpack_data** (*bool, optional*) – A flag indicating if the *LU_data* should be unpacked. If `False` , then the returned L and U are None. Default `True` .
- **unpack_pivots** (*bool, optional*) – A flag indicating if the *LU_pivots* should be unpacked into a permutation matrix P. If `False` , then the returned P is None. Default `True` .

Returns

Tuple of tensors(Pivots, L, U).

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> LU_data = mindspore.tensor([[[-0.3806, -0.4872,  0.5536],
...                              [-0.1287,  0.6508, -0.2396],
...                              [ 0.2583,  0.5239,  0.6902]],
...                             [[ 0.6706, -1.1782,  0.4574],
...                              [-0.6401, -0.4779,  0.6701],
...                              [ 0.1015, -0.5363,  0.6165]]])
>>> LU_pivots = mindspore.tensor(([[1, 3, 3], [2, 3, 3]]), mindspore.int32)
>>> pivots, L, U = mindspore.ops.lu_unpack(LU_data, LU_pivots)
>>> print(pivots)
[[[1. 0. 0.]
  [0. 0. 1.]
  [0. 1. 0.]]
 [[0. 0. 1.]
  [1. 0. 0.]
  [0. 1. 0.]]]
>>> print(L)
[[[ 1.       0.       0.]
  [-0.1287   1.       0.]
  [ 0.2583   0.5239   1.]]
 [[ 1.0000   0.       0.]
  [-0.6401   1.       0.]
  [ 0.1015  -0.5363   1.]]]
>>> print(U)
[[[-0.3806  -0.4872   0.5536]
  [ 0.       0.6508  -0.2396]
  [ 0.       0.       0.6902]]
 [[ 0.6706  -1.1782   0.4574]
  [ 0.      -0.4779   0.6701]
  [ 0.       0.       0.6165]]]
```

mindspore.ops.matmul

mindspore.ops.**matmul**(*input*, *other*)

Return the matrix product of two tensors.

Note:

- The dtype of *input* and *other* must be same.
- On Ascend, the rank of *input* or *other* must be between 1 and 6.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor or scalar

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> # case 1 : Reasonable application of broadcast mechanism.
>>> input = mindspore.ops.arange(24, dtype=mindspore.float32).reshape(2, 3, 4)
>>> other = mindspore.ops.arange(20, dtype=mindspore.float32).reshape(4, 5)
>>> output = mindspore.ops.matmul(input, other)
>>> print(output)
[[[  70,   76,   82,   88,   94],
  [ 190,  212,  234,  256,  278],
  [ 310,  348,  386,  424,  462]],
 [[ 430,  484,  538,  592,  646],
  [ 550,  620,  690,  760,  830],
  [ 670,  756,  842,  928, 1014]]]
>>>
>>> # case 2 : The rank of `input` is 1.
>>> input = mindspore.ops.ones(([1, 2]))
>>> other = mindspore.ops.ones(([2]))
>>> mindspore.ops.matmul(input, other)
Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00])
```

mindspore.ops.matrix_band_part

mindspore.ops.**matrix_band_part**(*x*, *lower*, *upper*)

Copy a tensor setting everything outside a central band in each innermost matrix to zero.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **x** (Tensor) – The input tensor.
- **lower** (*Union[int, Tensor]*) – Number of subdiagonals to keep. If negative, keep entire lower triangle.
- **upper** (*Union[int, Tensor]*) – Number of superdiagonals to keep. If negative, keep entire upper triangle.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.ops.ones([2, 4, 4])
>>> output = mindspore.ops.matrix_band_part(x, 2, 1)
>>> print(output)
[[[1. 1. 0. 0.]
  [1. 1. 1. 0.]
  [1. 1. 1. 1.]
  [0. 1. 1. 1.]]
 [[1. 1. 0. 0.]
  [1. 1. 1. 0.]
  [1. 1. 1. 1.]
  [0. 1. 1. 1.]]]
```

mindspore.ops.matrix_solve

mindspore.ops.**matrix_solve**(*matrix*, *rhs*, *adjoint=False*)

Solves systems of linear equations.

$$matrix[..., M, M] * x[..., M, K] = rhs[..., M, K]$$
$$adjoint(matrix[..., M, M]) * x[..., M, K] = rhs[..., M, K]$$

> **Warning:** On GPU, if the matrix is irreversible, an error may be reported or an unknown result may be returned.

Parameters

- **matrix** (Tensor) – The first input tensor.
- **rhs** (Tensor) – The second input tensor.

- **adjoint** (*bool, optional*) –Indicating whether to solve with matrix or its (block-wise) adjoint. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> matrix = mindspore.tensor([[5., 4.], [3., 1.]])
>>> rhs = mindspore.tensor([[7.], [2.]])
>>> result = mindspore.ops.matrix_solve(matrix, rhs)
>>> print(result)
[[0.14285707]
 [1.5714287 ]]
```

mindspore.ops.matrix_diag

`mindspore.ops.`**`matrix_diag`**(*x*, *k=0*, *num_rows=- 1*, *num_cols=- 1*, *padding_value=0*, *align='RIGHT_LEFT'*)

Return a tensor with the contents in *x* as k[0]-th to k[1]-th diagonals of a matrix, with everything else padded with *padding_value* .

num_rows and *num_cols* are tensors type of int32 with only one value, which is -1, indicating that the innermost matrix of the output tensor is a square.

Parameters

- **x** (`Tensor`) –The input tensor.
- **k** (*`Union[int, Tensor], optional`*) –Diagonal offsets. Positive value means superdiagonal, and negative value means subdiagonals. When *k* is a pair of integers specifying the low and high ends of a matrix band. Default `0` .
- **num_rows** (*`Union[int, Tensor], optional`*) –The number of rows of the output tensor. Default `-1` .
- **num_cols** (*`Union[int, Tensor], optional`*) –The number of columns of the output tensor. Default `-1` .
- **padding_value** (*`Union[int, float, Tensor], optional`*) –The number to fill the area outside the specified diagonal band. Default `0` .
- **align** (*`str, optional`*) –specifies how superdiagonals and subdiagonals should be aligned. Supported values `"RIGHT_LEFT"` , `"LEFT_RIGHT"` , `"LEFT_LEFT"` , `"RIGHT_RIGHT"` . Default `"RIGHT_LEFT"` .
  - When set to "RIGHT_LEFT", the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
  - When set to "LEFT_RIGHT", the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
  - When set to "LEFT_LEFT", the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
  - When set to "RIGHT_RIGHT", the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[8., 9., 0.],
...                       [1., 2., 3.],
...                       [0., 4., 5.]])
>>> k = mindspore.tensor([-1, 1], mindspore.int32)
>>> padding_value = mindspore.tensor(11.)
>>> num_rows = mindspore.tensor(3, mindspore.int32)
>>> num_cols = mindspore.tensor(3, mindspore.int32)
>>> output = mindspore.ops.matrix_diag(x, k, num_rows, num_cols, padding_value, align='LEFT_
↪RIGHT')
>>> print(output)
[[ 1.  8. 11.]
 [ 4.  2.  9.]
 [11.  5.  3.]]
>>> print(output.shape)
(3, 3)
```

mindspore.ops.matrix_diag_part

`mindspore.ops.`**`matrix_diag_part`**(*x*, *k*, *padding_value*, *align='RIGHT_LEFT'*)

Return a tensor that retains the values of the specified diagonal while setting all other elements to zero.

Input *k* and *padding_value* must be const tensor when taking graph mode.

Parameters

- **x** (`Tensor`) –The input tensor with rank r, where r >= 2.
- **k** (*`Union[int, Tensor], optional`*) –Diagonal offsets. Positive value means superdiagonal, and negative value means subdiagonals. When *k* is a pair of integers specifying the low and high ends of a matrix band.
- **padding_value** (`Tensor`) –The number to fill the area outside the specified diagonal band.
- **align** (*`str, optional`*) –specifies how superdiagonals and subdiagonals should be aligned. Supported values `"RIGHT_LEFT"` , `"LEFT_RIGHT"` , `"LEFT_LEFT"` , `"RIGHT_RIGHT"` . Default `"RIGHT_LEFT"` .
  - When set to "RIGHT_LEFT", the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
  - When set to "LEFT_RIGHT", the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
  - When set to "LEFT_LEFT", the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
  - When set to "RIGHT_RIGHT", the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1., 2., 3., 4.],
...                       [5., 6., 7., 8.],
...                       [9., 8., 7., 6.]])
>>> k = mindspore.tensor([1, 3], mindspore.int32)
>>> output = mindspore.ops.matrix_diag_part(x, k, mindspore.tensor(9.), align='RIGHT_LEFT')
>>> print(output)
[[9. 9. 4.]
 [9. 3. 8.]
 [2. 7. 6.]]
>>> print(output.shape)
(3, 3)
```

mindspore.ops.matrix_set_diag

mindspore.ops.**matrix_set_diag**(*x*, *diagonal*, *k=0*, *align='RIGHT_LEFT'*)

Return a tensor by replacing the elements on the k[0]-th to k[1]-th diagonals of the matrix *x* with the values from the input *diagonal* .

Parameters

- **x** (Tensor) −The input tensor with rank r, where r >= 2.
- **diagonal** (Tensor) −A diagonal tensor.
- **k** (*Union[int, Tensor], optional*) −Diagonal offsets. Positive value means superdiagonal, and negative value means subdiagonals. When *k* is a pair of integers specifying the low and high ends of a matrix band. Default 0 .
- **align** (*str, optional*) −specifies how superdiagonals and subdiagonals should be aligned. Supported values "RIGHT_LEFT" , "LEFT_RIGHT" , "LEFT_LEFT" , "RIGHT_RIGHT" . Default "RIGHT_LEFT" .
  - When set to "RIGHT_LEFT", the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
  - When set to "LEFT_RIGHT", the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
  - When set to "LEFT_LEFT", the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
  - When set to "RIGHT_RIGHT", the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[7., 7., 7., 7.],
...                       [7., 7., 7., 7.],
...                       [7., 7., 7., 7.]])
>>> diagonal = mindspore.tensor([[0., 9., 1.],
...                     [6., 5., 8.],
...                     [1., 2., 3.],
...                     [4., 5., 0.]])
>>> k = mindspore.tensor(([-1, 2]), mindspore.int32)
>>> align = 'RIGHT_LEFT'
>>> output = ops.matrix_set_diag(x, diagonal, k, align)
>>> print(output)
[[1. 6. 9. 7.]
 [4. 2. 5. 1.]
 [7. 5. 3. 8.]]
>>> print(output.shape)
(3, 4)
```

mindspore.ops.mm

mindspore.ops.**mm**(*input*, *mat2*)

Returns the matrix product of two arrays.

If *input* is a $(n \times m)$ tensor, *mat2* is a $(m \times p)$ tensor, *out* will be a $(n \times p)$ tensor.

Note:

- This function cannot support broadcasting. Refer to *mindspore.ops.matmul()* instead if you need a broadcastable function.
- On Ascend, float64 doesn't be supported.

Parameters

- **input** (Tensor) –The first matrix of matrix multiplication.
- **mat2** (Tensor) –The second matrix of matrix multiplication.

Returns

Tensor or scalar

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> out = mindspore.ops.mm(mindspore.ops.ones((2, 3)), mindspore.ops.ones((3, 4)))
>>> print(out.shape)
(2, 4)
```

mindspore.ops.mv

mindspore.ops.**mv**(*mat*, *vec*)

Multiplies matrix *mat* and vector *vec*.

If *mat* is a (N, M) tensor, *vec* is a 1-D M tensor, out will be a 1-D N tensor.

Parameters

- **mat** (Tensor) −Input matrix.
- **vec** (Tensor) −Input vector.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.mv(mindspore.tensor([[3., 4.], [1., 6.], [1., 3.]]), mindspore.
↪tensor([1., 2.]))
>>> print(output)
[11. 13. 7.]
```

mindspore.ops.outer

mindspore.ops.**outer**(*input*, *vec2*)

Compute outer product of two tensors.

If *input*' s length is n and *vec2*' s length is m , then output must be a matrix of shape (n, m) .

Note: This function does not broadcast.

Parameters

- **input** (Tensor) −The 1-D input tensor.
- **vec2** (Tensor) −The 1-D input tensor.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3])
>>> vec2 = mindspore.tensor([4, 5, 6])
>>> mindspore.ops.outer(input, vec2)
Tensor(shape=[3, 3], dtype=Int64, value=
[[ 4,  5,  6],
 [ 8, 10, 12],
 [12, 15, 18]])
```

mindspore.ops.orgqr

`mindspore.ops.`**`orgqr`**(*input*, *input2*)

Compute the first N columns of a product of Householder matrices.

Usually used to calculate the explicit representation of the orthogonal matrix Q returned by *mindspore.ops.Geqrf*.

Take the case of input without batch dimension as an example: Suppose input *input* is a matrix of size (M, N) after householder transformation. When the diagonal of *input* is set to 1, every column of lower triangular in *input* is denoted as w_j for j for $j = 1, \ldots, M$, this function returns the first N columns of the matrix

$$H_1 H_2 \ldots H_k \quad \text{with} \quad H_j = \mathrm{I}_M - \tau_j w_j w_j^{\mathrm{H}}$$

where I_M is the M-dimensional identity matrix. And when w is complex, w^{H} is the conjugate transpose, otherwise the transpose. The output matrix is the same size as the input matrix *input*. tau is corresponding to *input2*.

Parameters

- **input** (Tensor) −2-D or 3-D input tensor, householder vectors, shape $(*, M, N)$.
- **input2** (Tensor) −1-D or 2-D input tensor, householder reflection coefficients, shape $(*, K)$, where K is less than or equal to N, indicating.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-2.0, -1.0], [1.0, 2.0]])
>>> y, tau = mindspore.ops.geqrf(input)
>>> mindspore.ops.orgqr(y, tau)
Tensor(shape=[2, 2], dtype=Float32, value=
[[-8.94427061e-01,  4.47213590e-01],
 [ 4.47213590e-01,  8.94427180e-01]])
```

mindspore.ops.ormqr

mindspore.ops.**ormqr**(*input*, *tau*, *other*, *left=True*, *transpose=False*)

Calculates the product of a matrix *other* and a matrix Q (represented by Householder vectors *input* and Householder reflection coefficients *tau*).

If *left* is True, computes Q * *other* , otherwise, compute *other* * Q.

Parameters

- **input** (Tensor) –The input tensor, shape $(*, mn, k)$, when *left* is True, mn equals to m, otherwise, mn equals to n.
- **tau** (Tensor) –The input tensor, shape $(*, min(mn, k))$.
- **other** (Tensor) –The input tensor, shape $(*, m, n)$.
- **left** (*bool, optional*) –The order of computation. Default `True` .
- **transpose** (*bool, optional*) –Whether the matrix Q is conjugate transposed or not. Default `False` .

Returns

Tensor

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(([[-114.6, 10.9, 1.1], [-0.304, 38.07, 69.38], [-0.45, -0.17,␣
↪62]]),
...                           mindspore.float32)
>>> tau = mindspore.tensor(([1.55, 1.94, 3.0]), mindspore.float32)
>>> other = mindspore.tensor(([[-114.6, 10.9, 1.1],
...                            [-0.304, 38.07, 69.38],
...                            [-0.45, -0.17, 62]]), mindspore.float32)
>>> output = mindspore.ops.ormqr(input, tau, other)
>>> print(output)
[[  63.82713   -13.823125 -116.28614 ]
 [ -53.659264  -28.157839  -70.42702 ]
 [ -79.54292    24.00183   -41.34253 ]]
```

mindspore.ops.pinv

mindspore.ops.**pinv**(*x*, ***, *atol=None*, *rtol=None*, *hermitian=False*)

Computes the (Moore-Penrose) pseudo-inverse of a matrix.

This function is computed using SVD. If $x = U * S * V^T$,Than the pseudo-inverse of x is: $x^+ = V * S^+ * U^T$, S^+ is the reciprocal of each non-zero element on the diagonal of S, and zero remains in place. Batch matrices are supported. If x is a batch matrix, the output has the same batch dimension when atol or rtol is float. If atol or rtol is a Tensor, its shape must be broadcast to the singular value returned by x.svd . If x.shape is (B, M, N), and the shape of atol or rtol is (K, B), the output shape is (K, B, N, M). When the Hermitian is True, temporary support only real domain, x is treated as a real symmetric, so x is not checked internally, and only use the lower triangular part in the computations. When the singular value of x (or the norm of the eigenvalues when hermitian = True) that are below threshold $(max(atol, \sigma \cdot rtol)$, σ as the largest singular value or characteristic value), it is set to zero, and is not used in the computations. If rtol is not specified and x is a matrix of dimensions (M, N), then rtol is set to be

$rtol = max(M, N) * \varepsilon$, ε is the eps value of x.dtype. If rtol is not specified and atol specifies a value larger than zero, rtol is set to zero.

Note: This function uses svd internally, (or eigh , when *hermitian* = *True*). So it has the same problem as these functions. For details, see the warnings in svd() and eigh().

Parameters

x (Tensor) –The input tensor whose shape is $(*, M, N)$, * is zero or more batch dimensions. When *hermitian* is `True`, batch dimensions are not supported temporarily.

Keyword Arguments

- **atol** (float, Tensor) –The absolute tolerance value. Default `None` .
- **rtol** (float, Tensor) –The relative tolerance value. Default `None` .
- **hermitian** (bool) –Whether *x* is assumed to be symmetric if real. Default `False` .

Outputs:

A tensor whose shape is $(*, N, M)$, * is zero or more batch dimensions.

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[4., 0.], [0., 5.]], mindspore.float32)
>>> output = mindspore.ops.pinv(x)
>>> print(output)
[[0.25 0.  ]
 [0.   0.2 ]]
```

mindspore.ops.svd

mindspore.ops.**svd**(*input*, *full_matrices=False*, *compute_uv=True*)

Computes the singular value decompositions of one or more matrices.

If A is a matrix, the svd returns the singular values S, the left singular vectors U and the right singular vectors V. It meets:

$$A = U * diag(S) * V^T$$

Parameters

- **input** (Tensor) –The input tensor, shape is $(*, M, N)$.
- **full_matrices** (*bool, optional*) –If `True` , compute full-sized U and V. If `False`, compute only the leading P singular vectors, with P is the minimum of M and N. Default `False` .
- **compute_uv** (*bool, optional*) –If `True` , compute the left and right singular vectors. If `False`, compute only the singular values. Default `True` .

Returns

If compute_uv is `True` , a tuple(*s* , *u* , *v*) of tensors will be returned. Otherwise, only a single tensor -> *s* will be returned.

- *s* is the singular value tensor. The shape is $(*, P)$.
- *u* is the left singular tensor. If *compute_uv* is `False` , *u* will not be returned. The shape is $(*, M, P)$. If *full_matrices* is `True` , the shape will be $(*, M, M)$.
- *v* is the right singular tensor. If *compute_uv* is `False` , *v* will not be returned. The shape is $(*, N, P)$. If *full_matrices* is `True` , the shape will be $(*, N, N)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2], [-4, -5], [2, 1]], mindspore.float32)
>>> s, u, v = mindspore.ops.svd(input, full_matrices=True, compute_uv=True)
>>> print(s)
[7.0652843 1.040081 ]
>>> print(u)
[[ 0.30821905 -0.48819482  0.81649697]
 [-0.90613353  0.11070572  0.40824813]
 [ 0.2896955   0.8656849   0.4082479 ]]
>>> print(v)
[[ 0.63863593  0.769509  ]
 [ 0.769509   -0.63863593]]
```

mindspore.ops.slogdet

`mindspore.ops.`**`slogdet`**(*input*)

Computes the sign and the log of the absolute value of the determinant of one or more square matrices.

Note: The type of output always be real-value, even *input* is complex.

Parameters

`input` (`Tensor`) −The input tensor, shape is $(..., M, M)$.

Returns

Tuple of 2 tensors which are sign and the log of the absolute value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1., 2], [3, 4]])
>>> sign, value = mindspore.ops.slogdet(input)
>>> print(sign)
-1.0
>>> print(value)
0.6931472
>>> input = mindspore.tensor([[[-4.5, -1.5], [7.0, 6.0]], [[2.5, 0.5], [3.0, 9.0]]])
>>> sign, value = mindspore.ops.slogdet(input)
>>> print(sign)
[-1.  1.]
>>> print(value)
[2.8033605 3.0445223]
```

mindspore.ops.trace

mindspore.ops.**trace**(*input*)

Return the sum of the elements along the diagonal of the input tensor.

Note: Input must be matrix, and complex number is not supported at present.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[0, 1, 2],
...                           [3, 4, 5],
...                           [6, 7, 8]])
>>> mindspore.ops.trace(input)
Tensor(shape=[], dtype=Int64, value= 12)
```

mindspore.ops.tensor_dot

mindspore.ops.**tensor_dot**(*x1*, *x2*, *axes*)

Compute the tensor dot product along the specified axes.

Parameters

- **x1** (Tensor) –Input tensor.
- **x2** (Tensor) –Input tensor.
- **axes** (*Union[int, tuple(int), tuple(tuple(int)), list(list(int))]*) –The number of dimensions to sum over. If an integer *k* is provided, then sum over the last *k* axes of *x1* and the first *k* axes of *x2*, in order. If a tuple or list is provided, then *axes[0]* specifies the axes of *x1* and *axes[1]* specifies the axes of *x2*.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor, ops
>>> import mindspore
>>> import numpy as np
>>> input_x1 = Tensor(np.ones(shape=[1, 2, 3]), mindspore.float32)
>>> input_x2 = Tensor(np.ones(shape=[3, 1, 2]), mindspore.float32)
>>> output = ops.tensor_dot(input_x1, input_x2, ((0,1),(1,2)))
>>> print(output)
[[2. 2. 2]
 [2. 2. 2]
 [2. 2. 2]]
```

mindspore.ops.vander

mindspore.ops.**vander**(*x*, *N=None*)

Generates a Vandermonde matrix. The columns of the output matrix are powers of the input vector. The i-th output column is the input vector raised element-wise to the power of $N - i - 1$.

Parameters

- **x** (Tensor) –The 1-D input tensor.
- **N** (*int, optional*) –Number of columns in the output. Default None, *N* will be assigned as $len(x)$.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1., 2., 3., 5.])
>>> output = mindspore.ops.vander(input)
>>> print(output)
[[  1.   1.   1.   1.]
 [  8.   4.   2.   1.]
 [ 27.   9.   3.   1.]
 [125.  25.   5.   1.]]
>>> output = mindspore.ops.vander(input, N=3)
>>> print(output)
[[ 1.  1.  1.]
 [ 4.  2.  1.]
 [ 9.  3.  1.]
 [25.  5.  1.]]
```

mindspore.ops.vecdot

mindspore.ops.**vecdot**(*x*, *y*, ***, *axis=- 1*)

Calculates the dot product of two batches of vectors along the specified dimension.

Support broadcasting.

The formula of calculation is as follows. $\bar{x_i}$ represents the conjugate for complex vectors, and $\bar{x_i}$ is the raw value for real vectors.

$$\sum_{i=1}^{n} \bar{x_i} y_i$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **x** (Tensor) −The first batch of tensors.
- **y** (Tensor) −The second batch of tensors.

Keyword Arguments

axis (int) −Specify the axis for computation. Default `-1` .

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Note: Currently, complex numbers are not supported on GPU.

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[1, 3], [5, 7], [9, 8]])
>>> y = mindspore.tensor([[4, 5], [6, 7], [3, 2]])
>>> mindspore.ops.vecdot(x, y)
Tensor(shape=[3], dtype=Int64, value= [19, 79, 43])
>>> mindspore.ops.vecdot(x, y, axis=0)
Tensor(shape=[2], dtype=Int64, value= [61, 80])
```

2.2.5 Spectral Functions

API Name	Description	Supported Platforms
mindspore.ops.bartlett_window	Bartlett window function.	Ascend GPU CPU
mindspore.ops.blackman_window	Blackman window function.	Ascend GPU CPU
mindspore.ops.hamming_window	Hamming window function.	Ascend GPU CPU
mindspore.ops.hann_window	Hann window function.	Ascend GPU CPU
mindspore.ops.kaiser_window	Kaiser window function.	Ascend GPU CPU

mindspore.ops.bartlett_window

mindspore.ops.**bartlett_window**(*window_length*, *periodic=True*, *, *dtype=None*)

Bartlett window function.

A triangular-shaped weighting function used for smoothing or frequency analysis of signals in digital signal processing.

$$w[n] = 1 - \left|\frac{2n}{N-1} - 1\right| = \begin{cases} \frac{2n}{N-1} & \text{if } 0 \le n \le \frac{N-1}{2} \\ 2 - \frac{2n}{N-1} & \text{if } \frac{N-1}{2} < n < N \end{cases},$$

where N is the full window size, and n is natural number less than N :[0, 1, ⋯, N-1].

Parameters

- **window_length** (Tensor) –The size of window.
- **periodic** (*bool, optional*) –If True , return a periodic window. If False, return a symmetric window. Default True .

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default None .

Returns

A 1-D tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> window_length = mindspore.tensor(5)
>>> output = mindspore.ops.bartlett_window(window_length)
>>> print(output)
[0.  0.4 0.8 0.8 0.4]
>>> output = mindspore.ops.bartlett_window(window_length, periodic=False)
>>> print(output)
[0.  0.5 1.  0.5 0. ]
```

mindspore.ops.blackman_window

mindspore.ops.**blackman_window**(*window_length*, *periodic=True*, ***, *dtype=None*)

Blackman window function.

Usually used to extract finite signal segment for FFT.

$$w[n] = 0.42 - 0.5cos(\frac{2\pi n}{N-1}) + 0.08cos(\frac{4\pi n}{N-1})$$

where N is the full window size, and n is natural number less than N :[0, 1, ⋯, N-1].

Parameters

- **window_length** (Tensor) –The size of window.
- **periodic** (*bool, optional*) –If `True` , return a periodic window. If `False`, return a symmetric window. Default `True` .

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default `None` .

Returns

A 1-D tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> window_length = mindspore.tensor(10)
>>> output = mindspore.ops.blackman_window(window_length)
>>> print(output)
[-2.9802322e-08  4.0212840e-02  2.0077014e-01  5.0978714e-01
  8.4922993e-01  1.0000000e+00  8.4922981e-01  5.0978690e-01
  2.0077008e-01  4.0212870e-02]
```

mindspore.ops.hamming_window

mindspore.ops.**hamming_window**(*window_length*, *periodic=True*, *alpha=0.54*, *beta=0.46*, ***, *dtype=None*)

Hamming window function.

$$w[n] = \alpha\beta\cos\left(\frac{2\pi n}{N-1}\right),$$

where N is the full window size, and n is natural number less than N :[0, 1, ⋯, N-1].

Parameters

- **window_length** (*int*) –The size of window.
- **periodic** (*bool, optional*) –If `True` , return a periodic window. If `False`, return a symmetric window. Default `True` .
- **alpha** (*float, optional*) –The coefficient α. Default `0.54` .
- **beta** (*float, optional*) –The coefficient β. Default `0.46` .

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default `None` .

Returns

A 1-D tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> output = mindspore.ops.hamming_window(5)
>>> print(output)
[0.08000001 0.3978522  0.9121478  0.9121478  0.3978522 ]
>>> output = mindspore.ops.hamming_window(5, periodic=False)
>>> print(output)
[0.08000001 0.54       1.         0.54       0.08000001]
```

mindspore.ops.hann_window

mindspore.ops.**hann_window**(*window_length*, *periodic=True*, ***, *dtype=None*)

Hann window function.

$$w(n) = \frac{1}{2} - \frac{1}{2}\cos\left(\frac{2\pi n}{M-1}\right), \qquad 0 \le n \le M-1$$

Parameters

- **window_length** (*int*) –The size of window.
- **periodic** (*bool, optional*) –If `True` , return a periodic window. If `False`, return a symmetric window. Default `True` .

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default None .

Returns

A 1-D tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.hann_window(5)
>>> print(output)
[0.        0.3454915 0.9045085 0.9045085 0.3454915]
>>> output = mindspore.ops.hann_window(5, periodic=False)
>>> print(output)
[0.  0.5 1.  0.5 0. ]
```

mindspore.ops.kaiser_window

mindspore.ops.**kaiser_window**(*window_length*, *periodic=True*, *beta=12.0*, ***, *dtype=None*)

Kaiser window function.

$$w(n) = \frac{I_0\left(\beta\sqrt{1-\frac{4n^2}{(M-1)^2}}\right)}{I_0(\beta)}$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2}$$

where I_0 is the modified zeroth-order Bessel function.

Parameters

- **window_length** (*int*) –The size of window.
- **periodic** (*bool, optional*) –If True , return a periodic window. If False, return a symmetric window. Default True .
- **beta** (*float, optional*) –Shape parameter, when *beta* gets large, the window narrows. Default: 12.0 .

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type specified. Default None .

Returns

A 1-D tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> output = mindspore.ops.kaiser_window(5)
>>> print(output)
[5.27734413e-05 1.01719688e-01 7.92939834e-01 7.92939834e-01
 1.01719688e-01]
>>> output = mindspore.ops.kaiser_window(5, periodic=False)
>>> print(output)
[5.27734413e-05 2.15672745e-01 1.00000000e+00 2.15672745e-01
 5.27734413e-05]
```

2.3 Neural Network Layer Functions

2.3.1 Neural Network

API Name	Description	Supported Platforms
mindspore.ops.adaptive_avg_pool1d	Applies a 1D adaptive average pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.ops.adaptive_avg_pool2d	Performs 2D adaptive average pooling on a multi-plane input signal.	Ascend GPU CPU
mindspore.ops.adaptive_avg_pool3d	Performs 3D adaptive average pooling on a multi-plane input signal.	Ascend GPU CPU
mindspore.ops.adaptive_max_pool1d	Applies a 1D adaptive maximum pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.ops.adaptive_max_pool2d	This operator applies a 2D adaptive max pooling to an input signal composed of multiple input planes.	Ascend GPU CPU
mindspore.ops.avg_pool1d	Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.ops.avg_pool2d	Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.	Ascend GPU CPU
mindspore.ops.avg_pool3d	Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes.	Ascend GPU CPU
mindspore.ops.batch_norm	Batch Normalization for input data and updated parameters.	Ascend GPU CPU
mindspore.ops.bias_add	Returns the sum of the *input_x* and the *bias* Tensor.	Ascend GPU CPU
mindspore.ops.bidense	Applies bilinear dense connected layer for *input1* and *input2*.	Ascend GPU CPU
mindspore.ops.ctc_greedy_decoder	Performs greedy decoding on the logits given in inputs.	Ascend GPU CPU
mindspore.ops.conv1d	Applies a 1D convolution over an input tensor.	Ascend GPU
mindspore.ops.conv2d	Applies a 2D convolution over an input tensor.	Ascend GPU
mindspore.ops.conv3d	Applies a 3D convolution over an input tensor.	Ascend GPU

continues on next page

Table 11 – continued from previous page

mindspore.ops.deformable_conv2d	Given 4D tensor inputs *x*, *weight* and *offsets*, compute a 2D deformable convolution.	Ascend GPU CPU
mindspore.ops.dense	Applies the dense connected operation to the *input*.	Ascend GPU CPU
mindspore.ops.dropout	During training, randomly zeroes some of the elements of the input tensor with probability *p* from a Bernoulli distribution.	Ascend GPU CPU
mindspore.ops.dropout1d	During training, randomly zeroes some channels of the input tensor with probability p from a Bernoulli distribution(For a 3-dimensional tensor with a shape of NCL, the channel feature map refers to a 1-dimensional feature map with the shape of L).	Ascend GPU CPU
mindspore.ops.dropout2d	During training, randomly zeroes some channels of the input tensor with probability p from a Bernoulli distribution(For a 4-dimensional tensor with a shape of $NCHW$, the channel feature map refers to a 2-dimensional feature map with the shape of HW).	Ascend GPU CPU
mindspore.ops.dropout3d	During training, randomly zeroes some channels of the input tensor with probability p from a Bernoulli distribution(For a 5-dimensional tensor with a shape of $NCDHW$, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).	Ascend GPU CPU
mindspore.ops.embedding	Retrieve the word embeddings in *weight* using indices specified in *input*.	Ascend
mindspore.ops.flatten	Flatten a tensor along dimensions from *start_dim* to *start_dim*.	Ascend GPU CPU
mindspore.ops.fold	Combines an array of sliding local blocks into a large containing tensor.	Ascend GPU CPU
mindspore.ops.fractional_max_pool3d	Applies the 3D FractionalMaxPool operation over *input*.	GPU CPU
mindspore.ops.fused_infer_attention_score	This is a FlashAttention function designed for both incremental and full inference scenarios.	Ascend
mindspore.ops.speed_fusion_attention	The interface is used for self-attention fusion computing.	Ascend
mindspore.ops.group_norm	Group Normalization over a mini-batch of inputs.	Ascend GPU CPU
mindspore.ops.layer_norm	Applies the Layer Normalization on the mini-batch input.	Ascend
mindspore.ops.lp_pool1d	Applying 1D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.	Ascend GPU CPU
mindspore.ops.lp_pool2d	Applying 2D LPPooling operation on an input Tensor can be regarded as forming a 2D input plane.	Ascend GPU CPU
mindspore.ops.lrn	Local Response Normalization.	GPU CPU
mindspore.ops.max_pool2d	Performs a 2D max pooling on the input Tensor.	Ascend GPU CPU
mindspore.ops.max_pool3d	Performs a 3D max pooling on the input Tensor.	Ascend GPU CPU
mindspore.ops.max_unpool1d	Computes the inverse of *max_pool1d*.	Ascend GPU CPU
mindspore.ops.max_unpool2d	Computes the inverse of *max_pool2d*.	Ascend GPU CPU
mindspore.ops.max_unpool3d	Computes the inverse of mindspore.ops.max_pool3d().	Ascend GPU CPU
mindspore.ops.moe_token_permute	Permute the *tokens* based on the *indices*.	Ascend

continues on next page

Table 11 – continued from previous page

mindspore.ops.moe_token_unpermute	Unpermute a tensor of permuted tokens based on sorted indices, and optionally merge the tokens with their corresponding probabilities.	Ascend
mindspore.ops.incre_flash_attention	The interface for incremental inference.	Ascend
mindspore.ops.prompt_flash_attention	The interface for fully inference.	Ascend
mindspore.ops.flash_attention_score	Implement self-attention calculations in training scenarios.	Ascend
mindspore.ops.rms_norm	The RmsNorm(Root Mean Square Layer Normalization) operator is a normalization operation.	Ascend
mindspore.ops.unfold	Extracts sliding local blocks from a batched input tensor.	Ascend GPU CPU

mindspore.ops.adaptive_avg_pool1d

mindspore.ops.**adaptive_avg_pool1d**(*input*, *output_size*)

Applies a 1D adaptive average pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically, the input is of shape (N, C, L_{in}), adaptive_avg_pool1d outputs regional average in the L_{in}-dimension. The output is of shape (N, C, L_{out}), where L_{out} is defined by *output_size*.

Note: L_{in} must be divisible by *output_size*.

Parameters

- **input** (Tensor) –Tensor of shape (N, C, L_{in}), with float16 or float32 data type.
- **output_size** (*int*) –the target output size L_{out}.

Returns

Tensor of shape (N, C, L_{out}), has the same type as *input*.

Raises

- **TypeError** –If *output_size* is not an int.
- **TypeError** –If *input* is neither float16 nor float32.
- **ValueError** –If *output_size* is less than 1.
- **ValueError** –If length of shape of *input* is not equal to 3.
- **ValueError** –If the last dimension of *input* is smaller than *output_size*.
- **ValueError** –If the last dimension of *input* is not divisible by *output_size*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.random.randint(0, 10, [1, 3, 6]), mindspore.float32)
>>> output = ops.adaptive_avg_pool1d(input, output_size=2)
>>> print(output.shape)
(1, 3, 2)
```

mindspore.ops.adaptive_avg_pool2d

mindspore.ops.**adaptive_avg_pool2d**(*input*, *output_size*)

Performs 2D adaptive average pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is H x W. The number of output features is equal to the number of input features.

The input and output data format can be "NCHW" and "CHW". N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

For adaptive average pooling for 2D:

$$
\begin{aligned}
h_{start} &= floor(i * H_{in}/H_{out}) \\
h_{end} &= ceil((i+1) * H_{in}/H_{out}) \\
w_{start} &= floor(j * W_{in}/W_{out}) \\
w_{end} &= ceil((j+1) * W_{in}/W_{out}) \\
Output(i,j) &= \frac{\sum Input[h_{start}:h_{end}, w_{start}:w_{end}]}{(h_{end}-h_{start})*(w_{end}-w_{start})}
\end{aligned}
$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input of adaptive_avg_pool2d, which is a 3D or 4D tensor, with float16, float32 or float64 data type.
- **output_size** (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (H, W), or an int H for (H, H). H and W can be int or None. If it is None, it means the output size is the same as the input size.

Returns

Tensor, with the same type as the *input*.

Shape of the output is *input_shape[:len(input_shape) - len(out_shape)] + out_shape*.

$$
out\_shape = \begin{cases}
input\_shape[-2] + output\_size[1], & \text{if } output\_size \text{ is (None, w);} \\
output\_size[0] + input\_shape[-1], & \text{if } output\_size \text{ is (h, None);} \\
input\_shape[-2:], & \text{if } output\_size \text{ is (None, None);} \\
(h, h), & \text{if } output\_size \text{ is h;} \\
(h, w), & \text{if } output\_size \text{ is (h, w)}
\end{cases}
$$

Raises

- **ValueError** –If *output_size* is a tuple and the length of *output_size* is not 2.

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or float64.
- **ValueError** – If the dimension of *input* is less than or equal to the dimension of *output_size*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: output_size=(None, 2)
>>> input = Tensor(np.array([[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                          [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                          [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]),␣
 ↪mindspore.float32)
>>> output = ops.adaptive_avg_pool2d(input, (None, 2))
>>> print(output)
[[[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]]
>>> # case 2: output_size=2
>>> output = ops.adaptive_avg_pool2d(input, 2)
>>> print(output)
[[[3. 4.]
  [6. 7.]]
 [[3. 4.]
  [6. 7.]]
 [[3. 4.]
  [6. 7.]]]
>>> # case 3: output_size=(1, 2)
>>> output = ops.adaptive_avg_pool2d(input, (1, 2))
>>> print(output)
[[[4.5 5.5]]
 [[4.5 5.5]]
 [[4.5 5.5]]]
```

mindspore.ops.adaptive_avg_pool3d

mindspore.ops.**adaptive_avg_pool3d**(*input*, *output_size*)

Performs 3D adaptive average pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is (D, H, W). The number of output features is equal to the number of input planes.

Suppose the last 3 dimension size of x is (inD, inH, inW), the last 3 dimension size of output is $(outD, outH, outW)$.

$$
\begin{aligned}
&\forall \quad od \in [0, outD-1], oh \in [0, outH-1], ow \in [0, outW-1] \\
&output[od, oh, ow] = \\
&\qquad mean(x[istartD : iendD+1, istartH : iendH+1, istartW : iendW+1]) \\
&where, \\
&\qquad istartD = \left\lceil \frac{od * inD}{outD} \right\rceil \\
&\qquad iendD = \left\lfloor \frac{(od+1) * inD}{outD} \right\rfloor \\
&\qquad istartH = \left\lceil \frac{oh * inH}{outH} \right\rceil \\
&\qquad iendH = \left\lfloor \frac{(oh+1) * inH}{outH} \right\rfloor \\
&\qquad istartW = \left\lceil \frac{ow * inW}{outW} \right\rceil \\
&\qquad iendW = \left\lfloor \frac{(ow+1) * inW}{outW} \right\rfloor
\end{aligned}
$$

Parameters

- **input** (Tensor) –The input of adaptive_avg_pool3d, which is a 5D or 4D Tensor.
- **output_size** (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (D, H, W), or an int D for (D, D, D). D, H and W can be int or None which means the output size is the same as that of the input.

Returns

Tensor, with the same type as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **ValueError** –If the dimension of *input* is not 4D or 5D.
- **ValueError** –If *output_size* value is not positive.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: output_size=(3, 3, 4)
>>> output_size=(3, 3, 4)
>>> input_val = np.random.randn(4, 3, 5, 6, 7)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = ops.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(4, 3, 3, 3, 4)
```

(continues on next page)

(continued from previous page)

```
>>> # case 2: output_size=4
>>> output_size=5
>>> input_val = np.random.randn(2, 3, 8, 6, 12)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = ops.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(2, 3, 5, 5, 5)
>>> # case 3: output_size=(None, 4, 5)
>>> output_size=(None, 4, 5)
>>> input_val = np.random.randn(4, 1, 9, 10, 8)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = ops.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(4, 1, 9, 4, 5)
```

mindspore.ops.adaptive_max_pool1d

mindspore.ops.**adaptive_max_pool1d**(*input*, *output_size*)

Applies a 1D adaptive maximum pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically, the input is of shape (N, C, L_{in}), adaptive_max_pool1d outputs regional maximum in the L_{in}-dimension. The output is of shape (N, C, L_{out}), where L_{out} is defined by *output_size*.

Note:

- L_{in} must be divisible by *output_size*.
- Ascend platform only supports float16 type for input.

Parameters

- **input** (Tensor) –Tensor of shape (N, C, L_{in}), with float16 or float32 data type.
- **output_size** (*int*) –the target output size L_{out}.

Returns

Tensor of shape (N, C, L_{out}), has the same type as *input*.

Raises

- **TypeError** –If *input* is neither float16 nor float32.
- **TypeError** –If *output_size* is not an int.
- **ValueError** –If *output_size* is less than 1.
- **ValueError** –If the last dimension of *input* is smaller than *output_size*.
- **ValueError** –If the last dimension of *input* is not divisible by *output_size*.
- **ValueError** –If length of shape of *input* is not equal to 3.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.random.randint(0, 10, [1, 3, 6]), mindspore.float32)
>>> output = ops.adaptive_max_pool1d(input, output_size=2)
>>> print(output.shape)
(1, 3, 2)
```

mindspore.ops.adaptive_max_pool2d

mindspore.ops.**adaptive_max_pool2d**(*input*, *output_size*, *return_indices=False*)

This operator applies a 2D adaptive max pooling to an input signal composed of multiple input planes. That is, for any input size, the size of the specified output is H x W. The number of output features is equal to the number of input planes.

The input and output data format can be "NCHW" and "CHW". N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

$$
\begin{aligned}
h_{start} &= floor(i * H_{in}/H_{out}) \\
h_{end} &= ceil((i+1) * H_{in}/H_{out}) \\
w_{start} &= floor(j * W_{in}/W_{out}) \\
w_{end} &= ceil((j+1) * W_{in}/W_{out}) \\
Output(i,j) &= \max Input[h_{start}:h_{end}, w_{start}:w_{end}]
\end{aligned}
$$

Note: In KBK mode, *output_size* does not support mutable.

Parameters

- **input** (Tensor) −A 3D or 4D tensor, with float16, float32 or float64 data type.
- **output_size** (*Union[int, tuple]*) −The target output size. *output_size* can be a tuple (H, W), or an int H for (H, H). H and W can be int or None. If it is None, it means the output size is the same as the input size.
- **return_indices** (*bool, optional*) −If *return_indices* is True , the indices of max value would be output. Default: False .

Returns

Tensor, with the same shape and dtype as the *input*.

Raises

- **TypeError** −If *output_size* is not int or tuple.
- **TypeError** −If *input* is not a tensor.
- **TypeError** −If *return_indices* is not a bool.
- **TypeError** −If dtype of *input* is not float16, float32 or float64.
- **ValueError** −If *output_size* is a tuple and the length of *output_size* is not 2.
- **ValueError** −If the data format of *input* is not "NCHW" or "CHW".

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: output_size=(None, 2)
>>> input = Tensor(np.array([[[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                           [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                           [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]]),␣
↪mindspore.float32)
>>> output = ops.adaptive_max_pool2d(input, (None, 2))
>>> print(output)
[[[[2. 3.]
   [5. 6.]
   [8. 9.]]
  [[2. 3.]
   [5. 6.]
   [8. 9.]]
  [[2. 3.]
   [5. 6.]
   [8. 9.]]]]
>>> # case 2: output_size=2
>>> output = ops.adaptive_max_pool2d(input, 2)
>>> print(output)
[[[[5. 6.]
   [8. 9.]]
  [[5. 6.]
   [8. 9.]]
  [[5. 6.]
   [8. 9.]]]]
>>> # case 3: output_size=(1, 2)
>>> output = ops.adaptive_max_pool2d(input, (1, 2))
>>> print(output)
[[[[8. 9.]]
  [[8. 9.]]
  [[8. 9.]]]]
```

mindspore.ops.avg_pool1d

mindspore.ops.**avg_pool1d**(*input_x*, *kernel_size=1*, *stride=1*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically the input is of shape (N_{in}, C_{in}, L_{in}), avg_pool1d outputs regional average in the (L_{in})-dimension. Given kernel size $ks = l_{ker}$ and *stride* $s = s_0$, the operation is as follows.

$$\text{output}(N_i, C_j, l) = \frac{1}{l_{ker}} \sum_{n=0}^{l_{ker}-1} \text{input}(N_i, C_j, s_0 \times l + n)$$

Warning: *kernel_size* is in the range *[1, 255]*. *stride* is in the range *[1, 63]*.

Parameters

- **input_x** (Tensor) −Tensor of shape (N, C_{in}, L_{in}).

- **kernel_size** (*int, optional*) –The size of kernel window used to take the average value. Default: `1` .
- **stride** (*Union(int, tuple[int]), optional*) –The distance of kernel moving. *stride* can either be an int number or a tuple of one int number. Default: `1` .
- **padding** (*Union(int, tuple[int]), optional*) –The pad value to be filled. *padding* can either be an integer or a tuple of one integer. Default: `0` .
- **ceil_mode** (*bool, optional*) –If True, apply ceil instead of floor to compute the output shape. Default: `False`.
- **count_include_pad** (*bool, optional*) –If True, include the zero-padding in the averaging calculation. Default: `True` .

Returns

Tensor of shape (N, C_{out}, L_{out}).

Raises

- **TypeError** –If *input_x* is not a Tensor.
- **TypeError** –If *kernel_size* or *stride* is not an int.
- **TypeError** –If *ceil_mode* or *count_include_pad* is not a bool.
- **ValueError** –If length of shape of *input_x* is not equal to *3*.
- **ValueError** –If *kernel_size* or *stride* is less than *1*.
- **ValueError** –If *padding* is not int nor a tuple whose length is equal to *2*.
- **ValueError** –If value(s) of *padding* is less than *0*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.random.randint(0, 10, [1, 3, 6]), mindspore.float32)
>>> output = ops.avg_pool1d(input_x, kernel_size=6, stride=1)
>>> print(output.shape)
(1, 3, 1)
```

mindspore.ops.avg_pool2d

mindspore.ops.**avg_pool2d**(*input_x*, *kernel_size=1*, *stride=1*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=0*)

Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes. Typically the input is of shape $(N_{in}, C_{in}, H_{in}, W_{in})$, outputs regional average in the (H_{in}, W_{in})-dimension. Given kernel size (k_h, k_w) and *strides* , the operation is as follows.

$$\text{output}(N_i, C_j, h, w) = \frac{1}{k_h * k_w} \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \text{input}(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Warning: *kernel_size* is in the range *[1, 255]*. *stride* is in the range *[1, 63]*.

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **input_x** (Tensor) −Tensor of shape $(N, C_{in}, H_{in}, W_{in})$.
- **kernel_size** (*Union[int, tuple[int]]*) −The size of kernel used to take the average value. It is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively. Default: `1` .
- **stride** (*Union[int, tuple[int]]*) −The distance of kernel moving, an int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: `1` .
- **padding** (*Union(int, tuple[int])*) −The pad value to be filled. Default: 0. If *padding* is an integer, the paddings of top, bottom, left and right are the same, equal to pad. If *padding* is a tuple of *4* integers, the padding of top, bottom, left and right equal to *padding[0]*, *padding[1]*, *padding[2]* and *padding[3]* correspondingly. Default: `0` .
- **ceil_mode** (*bool*) −If True, apply ceil instead of floor to compute the output shape. Default: `False`.
- **count_include_pad** (*bool*) −If True, include the zero-padding in the averaging calculation. Default: `True` .
- **divisor_override** (*int*) −If specified, it will be used as divisor in the averaging calculation, otherwise *kernel_size* will be used. Default: `0`, which means not specified.

Returns

Tensor, with shape $(N, C_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** −If *input_x* is not a Tensor.
- **TypeError** −If *kernel_size* or *stride* is neither int nor tuple.
- **TypeError** −If *ceil_mode* or *count_include_pad* is not a bool.
- **TypeError** −If *divisor_override* is not an int.
- **ValueError** −If length of shape of *input_x* is not equal to *4*.
- **ValueError** −If *kernel_size* or *stride* is less than 1.
- **ValueError** −If *kernel_size* or *stride* is a tuple whose length is not equal to *2*.
- **ValueError** −If *padding* is not int nor a tuple whose length is equal to *4*.
- **ValueError** −If value(s) of *padding* is less than *0*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(1 * 3 * 3 * 4).reshape(1, 3, 3, 4), mindspore.float32)
>>> output = ops.avg_pool2d(x, kernel_size=2, stride=1)
>>> print(output)
[[[[ 2.5   3.5   4.5]
   [ 6.5   7.5   8.5]]
  [[14.5  15.5  16.5]
   [18.5  19.5  20.5]]
  [[26.5  27.5  28.5]
   [30.5  31.5  32.5]]]]
```

mindspore.ops.avg_pool3d

mindspore.ops.**avg_pool3d**(*input_x*, *kernel_size=1*, *stride=1*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=0*)

Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes. Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$, avg_pool3d outputs regional average in the (D_{in}, H_{in}, W_{in})-dimension. Given kernel size $ks = (d_{ker}, h_{ker}, w_{ker})$ and stride $s = (s_0, s_1, s_2)$, the operation is as follows.

$$\text{output}(N_i, C_j, d, h, w) = \frac{1}{d_{ker} * h_{ker} * w_{ker}} \sum_{l=0}^{d_{ker}-1} \sum_{m=0}^{h_{ker}-1} \sum_{n=0}^{w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Warning: *kernel_size* is in the range *[1, 255]*. *stride* is in the range *[1, 63]*.

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **input_x** (Tensor) –Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$. Currently support float16 and float32 data type.
- **kernel_size** (*Union[int, tuple[int]], optional*) –The size of kernel used to take the average value, is an int number that represents depth, height and width are both *kernel_size*, or a tuple of three int numbers that represent depth, height and width respectively. Default: `1` .
- **stride** (*Union[int, tuple[int]], optional*) –The distance of kernel moving, an int number that represents the depth, height and width of movement are both stride, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: `1` .
- **padding** (*Union(int, tuple[int]), optional*) –The pad value to be filled. If *padding* is an integer, the addings of head, tail, top, bottom, left and right are the same, equal to pad. If *padding* is a tuple of six integers, the padding of head, tail, top, bottom, left and right equal to padding[0], padding[1], padding[2], padding[3], padding[4] and padding[5] correspondingly. Default: `0` .
- **ceil_mode** (*bool, optional*) –If `True` , ceil instead of floor to compute the output shape. Default: `False` .
- **count_include_pad** (*bool, optional*) –If `True` , averaging calculation will include the zero-padding. Default: `True` .

- **divisor_override** (*int, optional*) –If specified, it will be used as divisor in the averaging calculation, otherwise *kernel_size* will be used. Default: `0` , which means not specified.

Returns

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$. Has the same data type with *input_x*.

Raises

- **TypeError** –If *input_x* is not a Tensor.
- **TypeError** –If *kernel_size*, *stride* or *padding* is neither an int not a tuple.
- **TypeError** –If *ceil_mode* or *count_include_pad* is not a bool.
- **TypeError** –If *divisor_override* is not an int.
- **ValueError** –If length of shape of *input_x* is not equal to *5*.
- **ValueError** –If numbers in *kernel_size* or *stride* are not positive.
- **ValueError** –If *kernel_size* or *stride* is a tuple whose length is not equal to *3*.
- **ValueError** –If *padding* is a tuple whose length is not equal to *6*.
- **ValueError** –If element of *padding* is less than *0*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.arange(1 * 2 * 2 * 2 * 3).reshape((1, 2, 2, 2, 3)), mindspore.
↪float16)
>>> output = ops.avg_pool3d(input_x, kernel_size=2, stride=1)
>>> print(output)
[[[[[ 5.  6.]]]
  [[[17. 18.]]]]]
```

mindspore.ops.batch_norm

mindspore.ops.**batch_norm**(*input_x*, *running_mean*, *running_var*, *weight*, *bias*, *training=False*, *momentum=0.1*, *eps=1e-5*)

Batch Normalization for input data and updated parameters.

Batch Normalization is widely used in convolutional neural networks. This operation applies Batch Normalization over inputs to avoid internal covariate shift as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the features using a mini-batch of data and the learned parameters can be described in the following formula,

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

where γ is *weight*, β is *bias*, ϵ is *eps*, $mean$ is the mean of x, $variance$ is the variance of x.

Warning:

- For Atlas 200/300/500 inference product, the result accuracy fails to reach 1‰ due to the square root instruction.

Note:

- If *training* is *False*, *weight*, *bias*, *running_mean* and *running_var* are Tensors.
- If *training* is *True*, *weight*, *bias*, *running_mean* and *running_var* are Parameters.

Parameters

- **input_x** (Tensor) −Tensor of shape (N, C), with float16 or float32 data type.
- **running_mean** (*Union[Tensor, Parameter]*) −The shape $(C,)$, has the same data type with *weight*.
- **running_var** (*Union[Tensor, Parameter]*) −The shape $(C,)$, has the same data type with *weight*.
- **weight** (*Union[Tensor, Parameter]*) −The shape $(C,)$, with float16 or float32 data type.
- **bias** (*Union[Tensor, Parameter]*) −The shape $(C,)$, has the same data type with *weight*.
- **training** (*bool, optional*) −If *training* is *True*, *mean* and *variance* are computed during training. If *training* is *False*, they're loaded from checkpoint during inference. Default: `False` .
- **momentum** (*float, optional*) −The hyper parameter to compute moving average for *running_mean* and *running_var* (e.g. $new\_running\_mean = (1 - momentum) * running\_mean + momentum * current\_mean$). Momentum value must be *[0, 1]*. Default: `0.1` .
- **eps** (*float, optional*) −A small value added for numerical stability. Default: `1e-5`, value must be *(0, 1]* .

Returns

output_x (Tensor) - The same type and shape as the *input_x*. The shape is (N, C).

Raises

- **TypeError** −If *training* is not a bool.
- **TypeError** −If dtype of *eps* or *momentum* is not float.
- **TypeError** −If *input_x*, *weight*, *bias*, *running_mean* or *running_var* is not a Tensor.
- **TypeError** −If dtype of *input_x*, *weight* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input_x = Tensor([[1.0, 2.0], [3.0, 4.0]], mindspore.float32)
>>> running_mean = Tensor([0.5, 1.5], mindspore.float32)
>>> running_var = Tensor([0.1, 0.2], mindspore.float32)
>>> weight = Tensor([2.0, 2.0], mindspore.float32)
>>> bias = Tensor([-1.0, -1.0], mindspore.float32)
>>> output = ops.batch_norm(input_x, running_mean, running_var, weight, bias)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[ 2.1621194  1.2360122]
 [14.810596  10.180061 ]]
```

mindspore.ops.bias_add

mindspore.ops.**bias_add**(*input_x*, *bias*)

Returns the sum of the *input_x* and the *bias* Tensor. Before adding, the *bias* Tensor will be broadcasted to be consistent with the shape of the *input_x* Tensor.

Parameters

- **input_x** (Tensor) –The input tensor. The shape can be 2-5 dimensions. Supported dtypes:
  - Ascend/CPU: all Number type.
  - GPU: float16, float32, int8.
- **bias** (Tensor) –The bias tensor, with shape (C). C must be the same as channel dimension C of *input_x*. It has the same type as *input_x*.

Returns

Tensor, with the same shape and data type as *input_x*.

Raises

- **TypeError** –If *input_x* or *bias* is not a Tensor.
- **TypeError** –If dtype of *input_x* and *bias* is inconsistent.
- **TypeError** –If dimension of *input_x* is not in the range [2, 5].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.arange(6).reshape((2, 3)), mindspore.float32)
>>> bias = Tensor(np.random.random(3).reshape((3)), mindspore.float32)
>>> output = ops.bias_add(input_x, bias)
>>> print(output.shape)
(2, 3)
```

mindspore.ops.bidense

mindspore.ops.**bidense**(*input1*, *input2*, *weight*, *bias=None*)

Applies bilinear dense connected layer for *input1* and *input2*. The bilinear dense function is defined as:

$$output = x_1^T A x_2 + b$$

x_1 represents *input1* , x_2 represents *input2* , A represents *weight* , b represents *bias* .

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input1** (Tensor) –Input Tensor of shape $(*, in1\_channels)$, where $*$ means any number of additional dimensions. All but the last dimension should be the same with *input2*.
- **input2** (Tensor) –Input Tensor of shape $(*, in2\_channels)$, where $*$ means any number of additional dimensions. All but the last dimension should be the same with *input1*.
- **weight** (Tensor) –The weight applied to the input1 and input2. The shape is $(out\_channels, in1\_channels, in2\_channels)$.
- **bias** (Tensor, *optional*) –Additive biases to the output. The shape is $(out\_channels)$ or (). Defaults: None , the *bias* is 0.

Returns

Tensor, shape $(*, out\_channels)$, where $*$ means any number of additional dimensions. All but the last dimension should be the same with the input Tensors.

Raises

- **TypeError** –If *input1* is not Tensor.
- **TypeError** –If *input2* is not Tensor.
- **TypeError** –If *weight* is not Tensor.
- **TypeError** –If *bias* is not Tensor.
- **ValueError** –If dimensions except the last of 'input1' are different from 'input2' .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input1 = mindspore.Tensor([[-1.1283, 1.2603],
...                            [0.0214, 0.7801],
...                            [-1.2086, 1.2849]], mindspore.float32)
>>> input2 = mindspore.Tensor([[-0.4631, 0.3238, 0.4201],
...                            [0.6215, -1.0910, -0.5757],
...                            [-0.7788, -0.0706, -0.7942]], mindspore.float32)
>>> weight = mindspore.Tensor([[[-0.3132, 0.9271, 1.1010],
...                             [0.6555, -1.2162, -0.2987]],
```

(continues on next page)

(continued from previous page)

```
...                             [[1.0458, 0.5886, 0.2523],
...                              [-1.3486, -0.8103, -0.2080]],
...                             [[1.1685, 0.5569, -0.3987],
...                              [-0.4265, -2.6295, 0.8535]],
...                             [[0.6948, -1.1288, -0.6978],
...                              [0.3511, 0.0609, -0.1122]]], mindspore.float32)
>>> output = ops.bidense(input1, input2, weight)
>>> print(output)
[[-2.0612743 0.5581219 0.22383511 0.8667302]
 [1.4476739 0.12626505 1.6552988 0.21297503]
 [0.6003161 2.912046 0.5590313 -0.35449564]]
```

mindspore.ops.ctc_greedy_decoder

mindspore.ops.**ctc_greedy_decoder**(*inputs*, *sequence_length*, *merge_repeated=True*)

Performs greedy decoding on the logits given in inputs.

Note: On Ascend, 'merge_repeated' can not be set to false.

Parameters

- **inputs** (Tensor) – The input Tensor must be a 3-D tensor whose shape is $(max\_time, batch\_size, num\_classes)$. *num_classes* must be *num_labels* + *1* classes, *num_labels* indicates the number of actual labels. Blank labels are reserved. Default blank label is *num_classes* - *1*. Data type must be float32 or float64.
- **sequence_length** (Tensor) – A tensor containing sequence lengths with the shape of $(batch\_size,)$. The type must be int32. Each value in the tensor must be equal to or less than *max_time*.
- **merge_repeated** (*bool*) – If `true` , merge repeated classes in output. Default: `True` .

Returns

decoded_indices (Tensor), A tensor with shape of $(total\_decoded\_outputs, 2)$. Data type is int64.

decoded_values (Tensor), A tensor with shape of $(total\_decoded\_outputs,)$, it stores the decoded classes. Data type is int64.

decoded_shape (Tensor), A tensor with shape of $(batch\_size, max\_decoded\_length)$. Data type is int64.

log_probability (Tensor), A tensor with shape of $(batch\_size, 1)$, containing sequence log-probability, has the same type as *inputs*.

Raises

- **TypeError** – If *merge_repeated* is not a bool.
- **ValueError** – If length of shape of *inputs* is not equal to 3.
- **ValueError** – If length of shape of *sequence_length* is not equal to 1.
- **ValueError** – If value in the *sequence_length* is larger than *max_time*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inputs = Tensor(np.array([[[0.6, 0.4, 0.2], [0.8, 0.6, 0.3]],
...                           [[0.0, 0.6, 0.0], [0.5, 0.4, 0.5]]]), mindspore.float32)
>>> sequence_length = Tensor(np.array([2, 2]), mindspore.int32)
>>> decoded_indices, decoded_values, decoded_shape, log_probability = ops.ctc_greedy_
↪decoder(inputs,
...                                                                                 ␣
↪sequence_length)
>>> print(decoded_indices)
[[0 0]
 [0 1]
 [1 0]]
>>> print(decoded_values)
[0 1 0]
>>> print(decoded_shape)
[2 2]
>>> print(log_probability)
[[-1.2]
 [-1.3]]
```

mindspore.ops.conv1d

mindspore.ops.**conv1d**(*input*, *weight*, *bias=None*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 1D convolution over an input tensor. The input Tensor is typically of shape (N, C_{in}, L_{in}), where N is batch size, C is channel number, L is input sequence width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, , $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, ranging from $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, ranging from $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size), where kernel_size is the width of the kernel. If we consider the input and output channels as well as the *groups* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{groups}, \text{kernel\_size})$, where *groups* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition and ConvNets .

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *groups>1*, condition $C_{in} = C_{out} = groups$ must be satisfied.

Parameters

- **input** (Tensor) −Input Tensor of shape (N, C_{in}, L_{in}).
- **weight** (Tensor) −The convolutional kernel value, it should has shape $(N, C_{in}/\text{groups}, \text{kernel\_size})$.
- **bias** (Tensor, *optional*) −Bias Tensor with shape (C_{out}). When bias is None, zeros will be used. Default: None .
- **stride** (*Union(int, tuple[int])*, *optional*) −The distance of kernel moving, an int number or a tuple of one int that represents width of movement. Default: 1.
- **pad_mode** (*str*, *optional*) −Specifies padding mode. The optional values are "same" , "valid" and "pad" . Default: "valid" .
  - "same": Adopts the way of completion. The height and width of the output will be equal to the input *x* divided by stride. The padding will be evenly calculated in left and right possiblily. Otherwise, the last extra padding will be calculated from the right side. If this mode is set, *padding* must be 0.
  - "valid": Adopts the way of discarding. The possible largest width of output will be returned without padding. Extra pixels will be discarded. If this mode is set, *padding* must be 0.
  - "pad": Implicit paddings on both sides of the input *x*. The number of *padding* will be padded to the input Tensor borders. *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int])*, *optional*) −Specifies the amount of padding to apply on both side of *input* when *pad_mode* is set to "pad". The paddings of left and right are the same, equal to padding or padding[0] when padding is a tuple of 1 integer. Default: 0 .
- **dilation** (*Union(int, tuple[int])*, *optional*) −Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 1 integer. Assuming $dilation = (d0,)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the width direction. The value should be in the ranges [1, L]. Default: 1 .
- **groups** (*int*, *optional*) −Splits *input* into groups. Default: 1 .

Returns

Tensor, the value that applied 1D convolution. The shape is (N, C_{out}, L_{out}). To see how different pad modes affect the output shape, please refer to *mindspore.nn.Conv1d* for more details.

Raises

- **TypeError** −If *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **TypeError** −*groups* is not an int.
- **TypeError** −If *bias* is not a Tensor.
- **ValueError** −If the shape of *bias* is not (C_{out}) .
- **ValueError** −If *stride* or *dilation* is less than 1.
- **ValueError** −If *pad_mode* is not one of 'same', 'valid' or 'pad'.
- **ValueError** −If *padding* is a tuple whose length is not equal to 1.
- **ValueError** −If *pad_mode* is not equal to 'pad' and *padding* is greater than 0.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(64).reshape((4, 4, 4)), mindspore.float32)
>>> weight = Tensor(np.arange(8).reshape((2, 2, 2)), mindspore.float32)
>>> bias = Tensor([-0.12345, 2.7683], mindspore.float32)
>>> output = ops.conv1d(x, weight, pad_mode='pad', padding=(1,), bias=bias, groups=2)
>>> print(output.shape)
(4, 2, 5)
```

mindspore.ops.conv2d

mindspore.ops.**conv2d**(*input*, *weight*, *bias=None*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 2D convolution over an input tensor. The input tensor is typically of shape $(N, C_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, H is feature height, W is feature width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, , $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{out_j})$ represents the bias of the j-th output channel, $weight(C_{out_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1]), where kernel_size[0] and kernel_size[1] are the height and width of the kernel, respectively. If we consider the input and output channels as well as the *groups* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{groups}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, where *groups* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition and ConvNets.

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *groups>1*, condition $C_{in} = C_{out} = groups$ must be satisfied.

Parameters

- **input** (Tensor) –Tensor of shape $(N, C_{in}, H_{in}, W_{in})$.
- **weight** (Tensor) –Tensor of shape $(N, C_{in}/\text{groups}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, then the size of kernel is (kernel_size[0], kernel_size[1]).
- **bias** (Tensor, *optional*) –Bias Tensor with shape (C_{out}). When bias is None , zeros will be used. Default: None .
- **stride** (*Union(int, tuple[int])*, *optional*) –The distance of kernel moving, an int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: 1 .
- **pad_mode** (*str*, *optional*) –Specifies padding mode. The optional values are "same" , "valid" and "pad" . Default: "valid" .
  - same: Adopts the way of completion. The height and width of the output will be equal to the input *x* divided by stride. The padding will be evenly calculated in top and bottom, left and right possiblily. Otherwise, the last extra padding will be calculated from the bottom and the right side. If this mode is set, *padding* must be 0.
  - valid: Adopts the way of discarding. The possible largest height and width of output will be returned without padding. Extra pixels will be discarded. If this mode is set, *padding* must be 0.
  - pad: Implicit paddings on both sides of the input *x*. The number of *padding* will be padded to the input Tensor borders. *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int])*, *optional*) –Implicit paddings on both sides of the input *x*. If *padding* is one integer, the paddings of top, bottom, left and right are the same, equal to padding. If *padding* is a tuple/list with 2 integers, the padding of top adn bottom is padding[0], and the padding of left and right is padding[1]. Default: 0 .
- **dilation** (*Union(int, tuple[int])*, *optional*) –Gaps between kernel elements.The data type is int or a tuple of 2 integers. Specifies the dilation rate to use for dilated convolution. If set to be $k > 1$, there will be $k - 1$ pixels skipped for each sampling location. Its value must be greater than or equal to 1 and bounded by the height and width of the input *x*. Default: 1 .
- **groups** (*int*, *optional*) –Splits *input* into groups. Default: 1 .

Returns

Tensor, the value that applied 2D convolution. The shape is $(N, C_{out}, H_{out}, W_{out})$. To see how different pad modes affect the output shape, please refer to *mindspore.nn.Conv2d* for more details.

Raises

- **TypeError** –If *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **TypeError** –*groups* is not an int.
- **TypeError** –If *bias* is not a Tensor.
- **ValueError** –If the shape of *bias* is not (C_{out}) .
- **ValueError** –If *stride* or *dilation* is less than 1.
- **ValueError** –If *pad_mode* is not one of 'same', 'valid' or 'pad'.
- **ValueError** –If *padding* is a tuple/list whose length is not equal to 2.
- **ValueError** –If *pad_mode* is not equal to 'pad' and *padding* is greater than 0.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> output = ops.conv2d(x, weight)
>>> print(output.shape)
(10, 32, 30, 30)
```

mindspore.ops.conv3d

mindspore.ops.**conv3d**(*input*, *weight*, *bias=None*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 3D convolution over an input tensor. The input tensor is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, D, H, W are the depth, height and width of the feature graph, respectively.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation , $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1], kernel_size[2]) where kernel_size[0] , kernel_size[1] and kernel_size[2] are the depth, height and width of the kernel, respectively. If we consider the input and output channels as well as the *groups* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{groups}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$, where *groups* is the number of groups dividing x's input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Note:

1. On Ascend platform, $groups = 1$ must be satisfied.
2. On Ascend platform, $dilation = 1$ must be satisfied.

Parameters

- **input** (Tensor) –Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$.

- **weight** (Tensor) −Set size of kernel is (kernel_size[0], kernel_size[1], kernel_size[2]), then the shape is $(C_{out}, C_{in}$, kernel_size[0], kernel_size[1], kernel_size[1]).
- **bias** (Tensor, *optional*) −Bias Tensor with shape (C_{out}). When bias is None, zeros will be used. Default: None .
- **stride** (*Union[int, tuple[int]], optional*) −The distance of kernel moving, it can be an int number that represents the depth, height and width of movement or a tuple of three int numbers that represent depth, height and width movement respectively. Default: 1 .
- **pad_mode** (*str, optional*) −Specifies padding mode. The optional values are "same" , "valid" and "pad" . Default: "valid" .
  - "same": Adopts the way of completion. The depth, height and width of the output will be equal to the input *x* divided by stride. The padding will be evenly calculated in head and tail, top and bottom, left and right directions possiblily. Otherwise, the last extra padding will be calculated from the tail, bottom and the right side. If this mode is set, *pad* must be 0.
  - "valid": Adopts the way of discarding. The possible largest depth, height and width of output will be returned without padding. Extra pixels will be discarded. If this mode is set, *pad* must be 0.
  - "pad": Implicit paddings on both sides of the input in depth, height and width. The number of *pad* will be padded to the input Tensor borders. *pad* must be greater than or equal to 0.
- **padding** (*Union[int, tuple[int], list[int]], optional*) −The pad value to be filled. If *pad* is an integer, the paddings of head, tail, top, bottom, left and right are the same, equal to pad. If *pad* is a tuple/list of 3 integers, the padding of head, tail, top, bottom, left and right equal to pad[0], pad[0], pad[1], pad[1], pad[2] and pad[2] correspondingly. Default: 0 .
- **dilation** (*Union[int, tuple[int]], optional*) −The data type is int or a tuple of 3 integers $(dilation_d, dilation_h, dilation_w)$. Currently, dilation on depth only supports the case of 1 on Ascend backend. Specifies the dilation rate to use for dilated convolution. If set $k > 1$, there will be $k - 1$ pixels skipped for each sampling location. The value ranges for the depth, height, and width dimensions are [1, D], [1, H], and [1, W], respectively. Default: 1 .
- **groups** (*int, optional*) −The number of groups into which the filter is divided. Default: 1 .

Returns

Tensor, the value that applied 3D convolution. The shape is $(N, C_{out}, D_{out}, H_{out}, W_{out})$. To see how different pad modes affect the output shape, please refer to *mindspore.nn.Conv3d* for more details.

Raises

- **TypeError** −If *out_channel* or *groups* is not an int.
- **TypeError** −If *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **TypeError** −If *bias* is not a Tensor.
- **ValueError** −If the shape of *bias* is not (C_{out}).
- **ValueError** −If *stride* or *dilation* is less than 1.
- **ValueError** −If *pad_mode* is not one of 'same', 'valid' or 'pad'.
- **ValueError** −If *padding* is a tuple or list whose length is not equal to 3.
- **ValueError** −If *pad_mode* is not equal to 'pad' and *pad* is greater than 0.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.ones([16, 3, 10, 32, 32]), mindspore.float16)
>>> weight = Tensor(np.ones([32, 3, 4, 3, 3]), mindspore.float16)
>>> output = ops.conv3d(x, weight, pad_mode="same", padding=0, stride=1, dilation=1,␣
↪groups=1)
>>> print(output.shape)
(16, 32, 10, 32, 32)
>>> output = ops.conv3d(x, weight, pad_mode="valid", padding=0, stride=1, dilation=1,␣
↪groups=1)
>>> print(output.shape)
(16, 32, 7, 30, 30)
>>> output = ops.conv3d(x, weight, pad_mode="pad", padding=(2, 1, 1), stride=1, dilation=1,␣
↪groups=1)
>>> print(output.shape)
(16, 32, 11, 32, 32)
```

mindspore.ops.deformable_conv2d

mindspore.ops.**deformable_conv2d**(*x*, *weight*, *offsets*, *kernel_size*, *strides*, *padding*, *bias=None*, *dilations=(1, 1, 1, 1)*, *groups=1*, *deformable_groups=1*, *modulated=True*)

Given 4D tensor inputs *x*, *weight* and *offsets*, compute a 2D deformable convolution. The deformable convolution operation can be expressed as follow:

Deformable Convolution v1:

$$y(p) = \sum_{k=1}^{K} w_k \cdot x(p + p_k + \Delta p_k)$$

Deformable Convolution v2:

$$y(p) = \sum_{k=1}^{K} w_k \cdot x(p + p_k + \Delta p_k) \cdot \Delta m_k$$

Where Δp_k and Δm_k are the learnable offset and modulation scalar for the k-th location. For details, please refer to Deformable ConvNets v2: More Deformable, Better Results and Deformable Convolutional Networks.

Parameters

- **x** (Tensor) −A 4D tensor of input image. With the format "NCHW", the shape is $(N, C_{in}, H_{in}, W_{in})$. Dtype: float16 or float32.
- **weight** (Tensor) −A 4D tensor of learnable filters. Must have the same type as *x*. The shape is $(C_{out}, C_{in}/groups, H_f, W_f)$.
- **offsets** (Tensor) −A 4D tensor of x-y coordinates offset and mask. With the format "NCHW", the shape is $(batch, 3 * deformable\_groups * H_f * W_f, H_{out}, W_{out})$. Note the C dimension is stored in the order of (offset_x, offset_y, mask). Must have the same type as *x*.
- **kernel_size** (*tuple[int]*) −A tuple of 2 integers. The size of kernel.
- **strides** (*tuple[int]*) −A tuple of 4 integers. The stride of the sliding window for each dimension of input. The dimension order is interpreted according to the data format of *x*. The N and C dimensions must be set to 1.
- **padding** (*tuple[int]*) −A tuple of 4 integers. The number of pixels to add to each (top, bottom, left, right) side of the input.

- **bias** (*Tensor, optional*) – An 1D tensor of additive biases to the filter outputs. The shape is (C_{out}). Default: `None`.
- **dilations** (*tuple[int], optional*) – A tuple of 4 integers. The dilation factor for each dimension of input. The dimension order is interpreted according to the data format of *x*. The N and C dimensions must be set to 1. Default: `(1, 1, 1, 1)`.
- **groups** (*int, optional*) – An integer of type int32. The number of blocked connections from input channels to output channels. In_channels and out_channels must both be divisible by *groups*. Default: `1`.
- **deformable_groups** (*int, optional*) – An integer of type int32. The number of deformable group partitions. In_channels must be divisible by *deformable_groups*. Default: `1`.
- **modulated** (*bool, optional*) – Specifies version of DeformableConv2D, True means v2, False means v1, currently only supports v2. Default: `True`.

Returns

Tensor, A 4D Tensor of output feature map. With the same type as *x*. With the format "NCHW", the shape is $(N, C_{out}, H_{out}, W_{out})$.

$$
\begin{aligned}
H_{out} &= \left\lfloor \frac{H_{in}+padding[0]+padding[1]-(H_f-1)\times \text{dilations}[2]-1}{\text{stride}[0]} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W_{in}+padding[2]+padding[3]-(W_f-1)\times \text{dilations}[3]-1}{\text{stride}[1]} + 1 \right\rfloor
\end{aligned}
$$

Raises

- **TypeError** – If *strides*, *padding*, *kernel_size* or *dilations* is not a tuple with integer elements.
- **TypeError** – If *modulated* is not a bool.
- **ValueError** – If the tuple size of *strides*, *padding*, *kernel_size* or *dilations* is not expected.
- **ValueError** – The N or C dimensions of *strides* or *dilations* is not set to 1.
- **ValueError** – If *modulated* is not set to True.

Warning: This is an experimental API that is subject to change or deletion.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.ones((4, 3, 10, 10)), mstype.float32)
>>> kh, kw = 3, 3
>>> weight = Tensor(np.ones((5, 3, kh, kw)), mstype.float32)
>>> offsets = Tensor(np.ones((4, 3 * kh * kw, 8, 8)), mstype.float32)
>>> output = ops.deformable_conv2d(x, weight, offsets, (kh, kw), (1, 1, 1, 1), (0, 0, 0, 0))
>>> print(output.shape)
(4, 5, 8, 8)
```

mindspore.ops.dense

mindspore.ops.**dense**(*input*, *weight*, *bias=None*)

Applies the dense connected operation to the *input*. The dense function is defined as:

$$output = input * weight^T + bias$$

Warning:

- This is an experimental API that is subject to change or deletion.
- On the Ascend platform, if *bias* is not 1D, the *input* cannot be greater than 6D in PYNATIVE or KBK mode.

Parameters

- **input** (Tensor) –Input Tensor of shape $(*, in\_channels)$, where $*$ means any number of additional dimensions.
- **weight** (Tensor) –The weight applied to the input. The shape is $(out\_channels, in\_channels)$ or $(in\_channels)$.
- **bias** (Tensor, *optional*) –Additive biases to the output. The shape is $(out\_channels)$ or (). Defaults: None, the *bias* is 0.

Returns

Output whose shape is determined by the shape of the input and the weight.

Raises

- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *weight* is not Tensor.
- **TypeError** –If *bias* is not Tensor.
- **RuntimeError** –On the Ascend platform, if *bias* is not 1D and *input* is greater than 6D in PYNATIVE or KBK mode.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input = Tensor([[-1., 1., 2.], [-3., -3., 1.]], mindspore.float32)
>>> weight = Tensor([[-2., -2., -2.], [0., -1., 0.]], mindspore.float32)
>>> bias = Tensor([0., 1.], mindspore.float32)
>>> output = ops.dense(input, weight, bias)
>>> print(output)
[[-4.  0.]
 [10.  4.]]
```

mindspore.ops.dropout

mindspore.ops.**dropout**(*input*, *p=0.5*, *training=True*, *seed=None*)

During training, randomly zeroes some of the elements of the input tensor with probability *p* from a Bernoulli distribution. It plays the role of reducing neuron correlation and avoid overfitting. And the return will be multiplied by $\frac{1}{1-p}$ during training. During the reasoning, this operation returns the same Tensor as the *x*.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input** (Tensor) –The input Tensor of shape $(*, N)$, with data type of float16, float32 or float64.
- **p** (*float, optional*) –The dropping rate, between 0 and 1, e.g. p = 0.1, means dropping out 10% of input units. Default: 0.5 .
- **training** (*bool, optional*) –Apply dropout if is True. Default: True.
- **seed** (*int, optional*) –Seed is used as entropy source for Random number engines generating pseudo-random numbers. Default: None , which will be treated as 0 .

Returns

- **output** (Tensor) - Zeroed tensor, with the same shape and data type as *input*.

Raises

- **TypeError** –If *p* is not a float.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input = Tensor(((20, 16), (50, 50)), mindspore.float32)
>>> output = ops.dropout(input, p=0.5)
>>> print(output.shape)
(2, 2)
```

mindspore.ops.dropout1d

mindspore.ops.**dropout1d**(*input*, *p=0.5*, *training=True*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution(For a 3-dimensional tensor with a shape of NCL, the channel feature map refers to a 1-dimensional feature map with the shape of L).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *1D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call which based on Bernoulli distribution probability *p*.

The parper Dropout: A Simple Way to Prevent Neural Networks from Overfitting mentioned this technology, And it is proved that it can effectively reduce over fitting and prevent neuronal coadaptation. For more details, refer to Improving neural networks by preventing co-adaptation of feature detectors .

dropout1d can improve the independence between channel feature maps.

Parameters

- **input** (Tensor) −A tensor with shape (N, C, L) or (C, L), where N is the batch size, C is the number of channels, L is the feature length. The data type must be int8, int16, int32, int64, float16, float32 or float64.
- **p** (*float, optional*) −The dropping probability of a channel, between 0 and 1, e.g. $p = 0.8$, which means an 80% chance of clearing. Default: `0.5` .
- **training** (*bool, optional*) −Apply dropout if is True. Default: `True` .

Returns

Tensor, output, with the same shape and data type as *input*.

Raises

- **TypeError** −If *input* is not a Tensor.
- **TypeError** −If the data type of *p* is not float.
- **ValueError** −If *p* is out of the range *[0.0, 1.0]*.
- **ValueError** −If *input* shape is not *2D* or *3D*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.random.randn(4, 3), mindspore.float32)
>>> output = ops.dropout1d(input_x, 0.5)
>>> print(output.shape)
(4, 3)
```

mindspore.ops.dropout2d

mindspore.ops.**dropout2d**(*input*, *p=0.5*, *training=True*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution(For a 4-dimensional tensor with a shape of *NCHW*, the channel feature map refers to a 2-dimensional feature map with the shape of *HW*).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *2D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call which based on Bernoulli distribution probability *p*. The parper Dropout: A Simple Way to Prevent Neural Networks from Overfitting mentioned this technology, And it is proved that it can effectively reduce over fitting and prevent neuronal coadaptation. For more details, refer to Improving neural networks by preventing co-adaptation of feature detectors .

dropout2d can improve the independence between channel feature maps.

Parameters

- **input** (Tensor) −A *4D* tensor with shape (N, C, H, W), where N is the batch size, C is the number of channels, H is the feature height, and W is the feature width. The data type must be int8, int16, int32, int64, float16, float32 or float64.
- **p** (*float*) −The dropping probability of a channel. The range is [0.0, 1.0], e.g. $p = 0.8$, which means dropping out 80% of channels. Default: `0.5` .
- **training** (*bool*) −If *training* is True, applying dropout, otherwise, not applying. Default: `True` .

Returns

Tensor, output, with the same shape and data type as *input*.

Raises

- **TypeError** −If *input* is not a Tensor.
- **TypeError** −If dtype of *input* is not int8, int16, int32, int64, float16, float32 or float64.
- **TypeError** −If the data type of *p* is not float.
- **ValueError** −If *p* is out of the range *[0.0, 1.0]*.
- **ValueError** −If *input* shape is not *4D*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.ones([2, 1, 2, 3]), mindspore.float32)
>>> output = ops.dropout2d(input, 0.5)
>>> print(output.shape)
(2, 1, 2, 3)
```

mindspore.ops.dropout3d

`mindspore.ops.`**`dropout3d`**(*input*, *p=0.5*, *training=True*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution(For a 5-dimensional tensor with a shape of $NCDHW$, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *3D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call which based on Bernoulli distribution probability *p*.

dropout3d can improve the independence between channel feature maps.

Parameters

- **input** (Tensor) −A *5D* tensor with shape (N, C, D, H, W), where N is the batch size, C is the number of channels, D is the feature depth, H is the feature height, and W is the feature width. The data type must be int8, int16, int32, int64, float16, float32 or float64.
- **p** (*float*) −The dropping probability of a channel, between 0 and 1, e.g. $p = 0.8$, which means dropping out 80% of channels. Default: `0.5` .
- **training** (*bool*) −If *training* is True, applying dropout, otherwise, not applying. Default: `True` .

Returns

Tensor, output, with the same shape and data type as *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not int8, int16, int32, int64, float16, float32 or float64.
- **TypeError** –If the data type of *p* is not float.
- **ValueError** –If *p* is out of the range *[0.0, 1.0]*.
- **ValueError** –If *input* shape is not 5D.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.ones([2, 1, 2, 1, 2]), mindspore.float32)
>>> output = ops.dropout3d(input, 0.5)
>>> print(output.shape)
(2, 1, 2, 1, 2)
```

mindspore.ops.embedding

`mindspore.ops.`**`embedding`**(*input*, *weight*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*)

Retrieve the word embeddings in *weight* using indices specified in *input*.

> **Warning:** On Ascend, the behavior is unpredictable when the value of input is invalid.

Parameters

- **input** (`Tensor`) –The indices used to lookup in the *weight*. The data type must be mindspore.int32 or mindspore.int64, and the value should be in range *[0, weight.shape[0])*.
- **weight** (*`Union[Parameter, Tensor]`*) –The matrix where to lookup from. The shape must be 2D.
- **padding_idx** (*`int, optional`*) –If the value is not None, the corresponding row of *weight* will not be updated in training. The value should be in range *[-weight.shape[0], weight.shape[0])* if it's not `None`. Default `None`.
- **max_norm** (*`float, optional`*) –If not None, firstly get the p-norm result of the *weight* specified by *input* where p is specified by *norm_type*; if the result is larger then *max_norm*, update the *weight* with $\frac{max\_norm}{result+1e^{-7}}$ in-place. Default `None`.
- **norm_type** (*`float, optional`*) –Indicates the value of p in p-norm. Default `2.0`.
- **scale_grad_by_freq** (*`bool, optional`*) –If `True` the gradients will be scaled by the inverse of frequency of the index in *input*. Default `False`.

Returns

Tensor, has the same data type as *weight*, the shape is $(*input.shape, weight.shape[1])$.

Raises

- **ValueError** – If *padding_idx* is out of valid range.
- **ValueError** – If the shape of *weight* is invalid.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, ops
>>> input = Tensor([[1, 0, 1, 1], [0, 0, 1, 0]])
>>> weight = Parameter(np.random.randn(3, 3).astype(np.float32))
>>> output = ops.embedding(input, weight, max_norm=0.4)
>>> print(output)
[[[ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01]],
 [[ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01]]]
```

mindspore.ops.flatten

mindspore.ops.**flatten**(*input*, *order='C'*, ***, *start_dim=1*, *end_dim=- 1*)

Flatten a tensor along dimensions from *start_dim* to *start_dim*.

Parameters

- **input** (Tensor) – The input Tensor.
- **order** (*str, optional*) – Only `'C'` and `'F'` are supported. `'C'` means to flatten in row-major (C-style) order. `'F'` means to flatten in column-major (Fortran-style) order. Default: `'C'` .

Keyword Arguments

- **start_dim** (int, *optional*) – The first dimension to flatten. Default: `1` .
- **end_dim** (int, *optional*) – The last dimension to flatten. Default: `-1` .

Returns

Tensor. If no dimensions are flattened, returns the original *input*, otherwise return the flattened Tensor. If *input* is a 0-dimensional Tensor, a 1-dimensional Tensor will be returned.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *order* is not string type.
- **ValueError** – If *order* is string type, but not `'C'` or `'F'`.
- **TypeError** – If *start_dim* or *end_dim* is not int.

- **ValueError** – If *start_dim* is greater than *end_dim* after canonicalized.
- **ValueError** – If *start_dim* or *end_dim* is not in range of [-input.dim, input.dim-1].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones(shape=[1, 2, 3, 4]), mindspore.float32)
>>> output = ops.flatten(input_x)
>>> print(output.shape)
(1, 24)
```

mindspore.ops.fold

`mindspore.ops.`**`fold`**(*input*, *output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Combines an array of sliding local blocks into a large containing tensor.

Consider a batched input tensor of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where N is the batch dimension, $C \times \prod(\text{kernel\_size})$ is the total number of values within each block (a block has $\prod(\text{kernel\_size})$ spatial locations each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_{d} \left\lfloor \frac{\text{output\_size}[d] + 2 \times \text{padding}[d] - \text{dilations}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{strides}[d]} + 1 \right\rfloor,$$

where d is over all spatial dimensions.

Therefore, *output_size* is the spatial shape of the large containing tensor of the sliding local blocks.

The *dilation*, *padding* and *stride* arguments specify how the sliding blocks are retrieved.

Warning:

- The input must be a 3-dimensional Tensor with shape $(N, C \times \prod(\text{kernel\_size}), L)$.
- The output must be a 4-dimensional Tensor with shape $(N, C, output\_size[0], output\_size[1], ...)$.

Parameters

- **input** (`Tensor`) – 3-D Tensor, supported dtypes: float16, float32, float64, complex64 and complex128.
- **output_size** (`Tensor`) – 1D tensor with *2* elements of data type int.
- **kernel_size** (*`Union[int, tuple[int], list[int]]`*) – The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width. Must be specified.
- **dilation** (*`Union[int, tuple[int], list[int]], optional`*) – The size of the dilation, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **padding** (*`Union[int, tuple[int], list[int]], optional`*) – The size of the padding, should be two int for height and width. If type is int, it means that height equal with width. Default: `0` .

- **stride** (*Union[int, tuple[int], list[int]], optional*) – The size of the stride, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .

Returns

A Tensor, with same type as *input* . And its shape is as described above.

Raises

- **TypeError** – If *output_size*, *kernel_size*, *stride*, *dilation*, *padding* data type is not int, tuple or list.
- **ValueError** – If *output_size*, *kernel_size*, *dilation*, *stride* value is not greater than zero or elements number more than *2*.
- **ValueError** – If *padding* value is less than zero or elements number more than *2*.
- **ValueError** – If *input.shape[1] != kernel_size[0] * kernel_size[1]*
- **ValueError** – If *input.shape[2]* does not match the calculated number of sliding blocks.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(input_data=np.random.rand(16, 64, 25), dtype=mstype.float32)
>>> output_size = Tensor(input_data=[8, 8], dtype=mstype.int32)
>>> output = ops.fold(x, output_size, [2, 2], [2, 2], [2, 2], [2, 2])
>>> print(output.shape)
(16, 16, 8, 8)
```

mindspore.ops.fractional_max_pool3d

mindspore.ops.**fractional_max_pool3d**(*input*, *kernel_size*, *output_size=None*, *output_ratio=None*, *return_indices=False*, *_random_samples=None*)

Applies the 3D FractionalMaxPool operation over *input*. The output Tensor shape can be determined by either *output_size* or *output_ratio*, and the step size is determined by *_random_samples*. *output_size* will take effect when *output_size* and *output_ratio* are set at the same time. And *output_size* and *output_ratio* can not be `None` at the same time.

Refer to the paper Fractional MaxPooling by Ben Graham for more details.

The input and output data format can be "NCDHW". N is the batch size, C is the number of channels, D the feature depth, H is the feature height, and W is the feature width.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) – The input of FractionalMaxPool3d, which is a 4D or 5D tensor. Tensor of data type: float16, float32, double. Supported shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$.

- **kernel_size** (*Union[int, tuple[int]]*) −The size of kernel used to take the maximum value, is an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively. The value must be a positive integer.
- **output_size** (*Union[int, tuple[int]], optional*) −The shape of the target *output_size*, is an int number that represents depth, height and width, or a tuple of three int numbers that represent depth, height and width respectively. The value must be a positive integer. Default: `None` .
- **output_ratio** (*Union[float, tuple[float]], optional*) −The ratio of target output shape to input shape. Specifying the size of the output tensor by using a ratio of the input size. Data type: float16, float32, double, and value is between (0, 1). Default: `None` .
- **return_indices** (*bool, optional*) −Whether to return the indices of max value. Default: `False` .
- **_random_samples** (`Tensor, optional`) −The random step of fractional_max_pool3d, which is a 3D tensor. Tensor of data type: float16, float32, double, and value is between [0, 1). Supported shape $(N, C, 3)$ or $(1, C, 3)$. Default: `None`, the values of *_random_samples* will be randomly distributed using uniform distribution over an interval [0,1).

Returns

- **y** (Tensor) - A tensor, the output of FractionalMaxPool3d. Has the same data type with *input*. Has the shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$, where $(D_{out}, H_{out}, W_{out})$ = *output_size* or $(D_{out}, H_{out}, W_{out})$ = *output_ratio* * (D_{in}, H_{in}, W_{in}) .
- **argmax** (Tensor) - The indices along with the outputs, which is a Tensor, with the same shape as the *y* and int32 data type. It will output only when *return_indices* is True.

Raises

- **TypeError** −If *input* is not a 4D or 5D tensor.
- **TypeError** −If *_random_samples* is not a 3D tensor.
- **TypeError** −If data type of *input* is not float16, float32, double, int32, int64.
- **TypeError** −If dtype of *_random_samples* is not float16, float32, double.
- **TypeError** −If dtype of *argmax* is not int32, int64.
- **TypeError** −if _random_samples to have the different dtypes as input.
- **ValueError** −If *output_size* is a tuple and if *output_size* length is not 3.
- **ValueError** −If *kernel_size* is a tuple and if *kernel_size* length is not 3.
- **ValueError** −If numbers in *output_size* or *kernel_size* is not positive.
- **ValueError** −if *output_size* and *output_ratio* are None at the same time.
- **ValueError** −If the first dimension size of *input* and *_random_samples* is not equal.
- **ValueError** −If the second dimension size of *input* and *_random_samples* is not equal.
- **ValueError** −If the third dimension size of *_random_samples* is not 3.

Supported Platforms:

`GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
...            .reshape([1, 1, 2, 2, 4]), mstype.float32)
>>> _random_samples = Tensor(np.array([0.7, 0.7, 0.7]).reshape([1, 1, 3]), mstype.float32)
>>> output, argmax = ops.fractional_max_pool3d(x, kernel_size=(1, 1, 1), output_size=(1, 1,␣
↪3),
...                                            _random_samples=_random_samples, return_
↪indices=True)
>>> print(output)
[[[[[13. 14. 16.]]]]]
>>> print(argmax)
[[[[[12 13 15]]]]]
>>> output, argmax = ops.fractional_max_pool3d(x, kernel_size=(1, 1, 1), output_ratio=(0.5,␣
↪0.5, 0.5),
...                                            _random_samples=_random_samples, return_
↪indices=True)
>>> print(output)
[[[[[13. 16.]]]]]
>>> print(argmax)
[[[[[12 15]]]]]
```

mindspore.ops.fused_infer_attention_score

mindspore.ops.**fused_infer_attention_score**(*query*, *key*, *value*, ***, *pse_shift=None*, *atten_mask=None*, *actual_seq_lengths=None*, *actual_seq_lengths_kv=None*, *dequant_scale1=None*, *quant_scale1=None*, *dequant_scale2=None*, *quant_scale2=None*, *quant_offset2=None*, *antiquant_scale=None*, *antiquant_offset=None*, *key_antiquant_scale=None*, *key_antiquant_offset=None*, *value_antiquant_scale=None*, *value_antiquant_offset=None*, *block_table=None*, *query_padding_size=None*, *kv_padding_size=None*, *key_shared_prefix=None*, *value_shared_prefix=None*, *actual_shared_prefix_len=None*, *num_heads=1*, *scale=1.0*, *pre_tokens=2147483647*, *next_tokens=2147483647*, *input_layout='BSH'*, *num_key_value_heads=0*, *sparse_mode=0*, *inner_precise=1*, *block_size=0*, *antiquant_mode=0*, *key_antiquant_mode=0*, *value_antiquant_mode=0*, *softmax_lse_flag=False*)

This is a FlashAttention function designed for both incremental and full inference scenarios. It supports full inference scenarios (PromptFlashAttention) as well as incremental inference scenarios (IncreFlashAttention). When the S dimension of the query tensor (Q_S) equals 1, it enters the IncreFlashAttention branch; otherwise, it enters the PromptFlashAttention branch.

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d}})V$$

Warning:

- This is an experimental API that is subject to change or deletion.
- For Ascend, only the Atlas A2 training series products and Atlas 800I A2 inference products are currently supported.

Note:

- The data layout formats of query, key and value can be interpreted from multiple dimensions, as shown below:
  - B, Batch size. Represents the batch size of the input samples.
  - S, Sequence length. Represents the sequence length of the input samples. S1 represents the sequence length of the query, and S2 represents the sequence length of the key/value.
  - H, Head size. Represents the size of the hidden layer.
  - N, Head nums. Represents the number of attention heads.
  - D, Head dims. Represents the smallest unit size of the hidden layer, satisfying $D = H/N$.

Parameters

- **query** (Tensor) –The query input of the attention structure, with data type of float16, bfloat16 or int8. Input tensor of shape (B, S, H), (B, N, S, D), or (B, S, N, D).
- **key** (*Union[Tensor, tuple[Tensor], list[Tensor]]*) –The key input of the attention structure, with data type of float16, bfloat16 or int8. Input tensor of shape (B, S, H), (B, N, S, D), or (B, S, N, D).
- **value** (*Union[Tensor, tuple[Tensor], list[Tensor]]*) –The value input of the attention structure, with data type of float16, bfloat16 or int8. Input tensor of shape (B, S, H), (B, N, S, D), or (B, S, N, D).

Keyword Arguments

- **pse_shift** (Tensor, *optional*) –The padding mask tensor with data type of float16 or bfloat16. Default: None.
  - When Q_S is not 1, if pse_shift is of type float16, the query must be of type float16 or int8. If pse_shift is of type bfloat16, the query must also be of type bfloat16. The input shape must be either (B, N, Q_S, KV_S) or $(1, N, Q\_S, KV\_S)$, where Q_S corresponds to the S dimension of the query shape, and KV_S corresponds to the S dimension of the key and value shapes. For scenarios where the KV_S of pse_shift is not 32-aligned, it is recommended to pad it to 32 bytes to improve performance. The padding values for the extra portions are not restricted.
  - When Q_S is 1, if pse_shift is of type float16, the query must also be of type float16. If pse_shift is of type bfloat16, the query must be of type bfloat16. The input shape must be $(B, N, 1, KV\_S)$ or $(1, N, 1, KV\_S)$, where KV_S corresponds to the S dimension of the key/value shapes. For scenarios where the KV_S of pse_shift is not 32-aligned, it is recommended to pad it to 32 bytes to improve performance. The padding values for the extra portions are not restricted.
- **atten_mask** (Tensor, *optional*) –The attention mask tensor for the result of query*key with data type of int8, uint8 or bool. For each element, 0 indicates retention and 1 indicates discard. Default: None.
  - When Q_S is not 1, the recommended input shapes are Q_S,KV_S; B,Q_S,KV_S; 1,Q_S,KV_S; B,1,Q_S,KV_S or 1,1,Q_S,KV_S.
  - When Q_S is 1, the recommended input shapes are B,KV_S; B,1,KV_S or B,1,1,KV_S.
- **actual_seq_lengths** (*Union[tuple[int], list[int], Tensor], optional*) –Describe actual sequence length of the query with data type of int64. If this parameter is not specified, it can be set to None, indicating that it matches the S dimension of the query shape. Constraint: The effective sequence length for each batch in this parameter should not exceed the corresponding batch's sequence length in the query. When Q_S is 1, this parameter is ignored. Default: None.
- **actual_seq_lengths_kv** (*Union[tuple[int], list[int], Tensor], optional*) –Describe actual sequence length of the key and value with data type of int64. If this parameter is not specified, it can be set to None, indicating that it matches the S dimension of the key and value shape. Constraint: The effective sequence length

for each batch in this parameter should not exceed the corresponding batch's sequence length in the key and value. Default: `None`.

- **dequant_scale1** (`Tensor, optional`) –Quantization factors for inverse quantization after BMM1 with data type of uint64. Supports per-tensor mode. If not used, set it to None. Default: `None`.
- **quant_scale1** (`Tensor, optional`) –Quantization factors for quantization before BMM2 with data type of float32. Supports per-tensor mode. If not used, set it to None. Default: `None`.
- **dequant_scale2** (`Tensor, optional`) –Quantization factors for inverse quantization after BMM2 with data type of uint64. Supports per-tensor mode. If not used, set it to None. Default: `None`.
- **quant_scale2** (`Tensor, optional`) –Quantization factors for output quantization with data type of float32, bfloat16. Supports per-tensor and per-channel modes. If not used, set it to None. Default: `None`.
- **quant_offset2** (`Tensor, optional`) –Quantization offset for output quantization with data type of float32, bfloat16. Supports per-tensor and per-channel modes. If not used, set it to None. Default: `None`.

  For scenarios where the input is int8 and the output is int8: the parameters dequant_scale1, quant_scale1, dequant_scale2, and quant_scale2 must all be provided. The parameter quant_offset2 is optional and defaults to 0 if not specified.

  – When the output is int8 and quant_scale2 and quant_offset2 are per-channel, left padding, Ring Attention, or D-axis misalignment (not 32-aligned) scenarios are not supported.

  – When the output is int8, scenarios with sparse_mode as band and pre_tokens/next_tokens being negative are not supported.

  – When the output is int8, if quant_offset2 is not None and empty tensor, and the sparse_mode, pre_tokens, and next_tokens meet the following conditions, certain rows of the matrix may not participate in calculations, leading to errors. This scenario will be intercepted (solution: if this scenario should not be intercepted, quantization should be performed outside the FIA interface, not enabled inside the FIA interface):

    * sparse_mode = 0, if atten_mask is a not None and each batch's actual_seq_lengths - actual_seq_lengths_kv - pre_tokens > 0 or next_tokens < 0, it will meet the interception condition.
    * sparse_mode = 1 or 2, no interception condition will occur.
    * sparse_mode = 3, if each batch's actual_seq_lengths - actual_seq_lengths_kv < 0, it will meet the interception condition.
    * sparse_mode = 4, if pre_tokens < 0 or each batch's next_tokens + actual_seq_lengths - actual_seq_lengths_kv < 0, it will meet the interception condition.

  For scenarios where the input is int8 and the output is float16: the parameters dequant_scale1, quant_scale1, and dequant_scale2 must all be provided.

  For scenarios where the input is entirely float16 or bfloat16 and the output is int8: the parameter quant_scale2 must be provided. The parameter quant_offset2 is optional and defaults to 0 if not specified.

  The parameters quant_scale2 and quant_offset2 support both per-tensor and per-channel modes and two data types: float32 and bfloat16. If quant_offset2 is provided, its type and shape must match those of quant_scale2. When the input is bfloat16, both float32 and bfloat16 are supported; otherwise, only float32 is supported. For per-channel mode: When the output layout is BSH, the product of all dimensions in quant_scale2 must equal H. For other layouts, the product must equal N * D. When the output layout is BSH, it is recommended to set the shape of quant_scale2 as $(1, 1, H)$ or (H). When the output layout is BNSD, it is recommended to set the shape as $(1, N, 1, D)$ or (N, D). When the output layout is BSND, it is recommended to set the shape as $(1, 1, N, D)$ or (N, D).

- **antiquant_scale** (`Tensor, optional`) –Inverse quantization factors with data type of float16, float32 or bfloat16. Only support float16 when Q_S > 1. Supports per-tensor, per-channel and per-token modes. Default: `None`.
- **antiquant_offset** (`Tensor, optional`) –Inverse quantization offset with data type of float16, float32 or bfloat16. Only support float16 when Q_S > 1. Supports per-tensor, per-channel and per-token modes. Default: `None`.

Constraints for antiquant_scale and antiquant_offset parameters:

- Supports three modes: per-channel, per-tensor, and per-token:
  - Per-channel mode: The shape of both parameters in the BNSD layout is $(2, N, 1, D)$, the shape in the BSND layout is $(2, N, D)$, and the shape in the BSH layout is $(2, H)$, where 2 corresponds to the key and value, and N represents num_key_value_heads. The parameter data type is the same as the query data type, and antiquant_mode should be set to 0.
  - Per-tensor mode: The shape of both parameters is (2), the data type is the same as the query data type, and antiquant_mode should be set to 0.
  - Per-token mode: The shape of both parameters is $(2, B, S)$, the data type is fixed to float32, and antiquant_mode should be set to 1.
- Supports both symmetric and asymmetric quantization:
  - Asymmetric quantization mode: Both antiquant_scale and antiquant_offset must be provided.
  - Symmetric quantization mode: antiquant_offset can be empty (`None`). If antiquant_offset is empty, symmetric quantization is performed. If antiquant_offset is provided, asymmetric quantization is performed.

- **key_antiquant_scale** (*Tensor, optional*) – Inverse quantization factors for the key, with data type of float16, float32 or bfloat16, when the KV fake quantization parameters are separated. Supports per-tensor, per-channel and per-token modes. Default: `None`. Invalid when Q_S > 1.
- **key_antiquant_offset** (*Tensor, optional*) – Inverse quantization offset for the key, with data type of float16, float32 or bfloat16, when the KV fake quantization parameters are separated. Supports per-tensor, per-channel and per-token modes. Default: `None`. Invalid when Q_S > 1.
- **value_antiquant_scale** (*Tensor, optional*) – Inverse quantization factors for the value, with data type of float16, float32 or bfloat16, when the KV fake quantization parameters are separated. Supports per-tensor, per-channel and per-token modes. Default: `None`. Invalid when Q_S > 1.
- **value_antiquant_offset** (*Tensor, optional*) – Inverse quantization offset for the value, with data type of float16, float32 or bfloat16, when the KV fake quantization parameters are separated. Supports per-tensor, per-channel and per-token modes. Default: `None`. Invalid when Q_S > 1.
- **block_table** (*Tensor, optional*) – Block mapping table in KV cache for PageAttention, with data type of int32. If not used, set it to None. Default: `None`. Invalid when Q_S > 1.
- **query_padding_size** (*Tensor, optional*) – The query padding size with data type of int64. Indicates whether the data in each batch of the query is right-aligned, and how many elements are right-aligned. Default: `None`. Invalid when Q_S is 1.
- **kv_padding_size** (*Tensor, optional*) – The key and value padding size with data type of int64. Indicates whether the data in each batch of the key and value is right-aligned, and how many elements are right-aligned. Default: `None`. Invalid when Q_S is 1.
- **key_shared_prefix** (*Tensor, optional*) – Shared prefix of the key. This is a reserved parameter and is not yet enabled. Default: `None`.
- **value_shared_prefix** (*Tensor, optional*) – Shared prefix of the value. This is a reserved parameter and is not yet enabled. Default: `None`.
- **actual_shared_prefix_len** (*Union[tuple[int], list[int], Tensor], optional*) – Describe the actual length of shared prefix. This is a reserved parameter and is not yet enabled. Default: `None`.
- **num_heads** (*int, optional*) – The number of heads in the query, equal to N when input_layout is BNSD. Default: `1`.
- **scale** (*double, optional*) – The scale value indicating the scale coefficient, which serves as the scalar value for the Muls in the calculation. Generally, the value is $1.0/\sqrt{d}$. Default: `1.0`.

- **pre_tokens** (int, *optional*) −Parameter for sparse computation, represents how many tokens are counted forward. Default: 2147483647. Invalid when Q_S is 1.
- **next_tokens** (int, *optional*) −Parameter for sparse computation, represents how many tokens are counted backward. Default: 2147483647. Invalid when Q_S is 1.
- **input_layout** (*str*, *optional*) −Specifies the layout of input query, key and value. BSH, BNSD, BSND or BNSD_BSND is supported. When the layout is BNSD_BSND, it means the input is in the BNSD format and the output is in the BSND format, this is only supported when Q_S > 1. Default: BSH.
- **num_key_value_heads** (int, *optional*) −Head numbers of key/value which are used in GQA (Grouped-Query Attention) scenario. Default: 0. A value of 0 means it is equal to the number of key/value heads. The num_heads must be divisible by num_key_value_heads, and the ratio of num_heads to num_key_value_heads must not be greater than 64. When the layout is BNSD, the num_key_value_heads must also equals to the N dimension of the key/value shapes, otherwise, an execution error will occur.
- **sparse_mode** (int, *optional*) −Indicates sparse mode. Default 0. Invalid when Q_S is 1.
  - 0: Indicates the defaultMask mode. If atten_mask is not passed, the mask operation is not performed, and pre_tokens and next_tokens(internally assigned as INT_MAX) are ignored. If passed in, the complete atten_mask matrix (S1 * S2) also must be passed in, indicating that the part between pre_tokens and next_tokens needs to be calculated.
  - 1: Represents allMask. The complete atten_mask matrix (S1 * S2) is required.
  - 2: Represents the mask in leftUpCausal mode. The optimized atten_mask matrix (2048*2048) is required.
  - 3: Represents the mask in rightDownCausal mode, corresponding to the lower triangular scenario divided by the right vertex. The optimized atten_mask matrix (2048*2048) is required.
  - 4: Represents the mask in band mode, that is, the part between counting pre_tokens and next_tokens. The optimized atten_mask matrix (2048*2048) is required.
  - 5: Represents the prefix scenario, not implemented yet.
  - 6: Represents the global scenario, not implemented yet.
  - 7: Represents the dilated scenario, not implemented yet.
  - 8: Represents the block_local scenario, not implemented yet.
- **inner_precise** (int, *optional*) −There are four modes: 0, 1, 2, and 3, represented by 2 bits: bit 0 (bit0) represents the choice for high precision or high performance, and bit 1 (bit1) indicates whether row-wise invalidity correction is applied.
  - 0: Enable high-precise mode, without row-wise invalidity correction.
  - 1: High-performance mode, without row-wise invalidity correction.
  - 2: Enable high-precise mode, with row-wise invalidity correction.
  - 3: High-performance mode, with row-wise invalidity correction.

  When Q_S > 1, if sparse_mode is 0 or 1 and a user-defined mask is provided, it is recommended to enable row-wise invalidity correction. Only support 0 and 1 when Q_S is 1. Default: 1.

  High-precise and high-performance are only effective for float16 inputs; Row invalidity correction is effective for float16, bfloat16, and int8 inputs. Currently, 0 and 1 are reserved configuration values. If there is a situation where an entire row in the "mask portion involved in computation" is all 1s, precision may degrade. In such cases, you can try setting this parameter to 2 or 3 to enable row invalidity correction for improved precision. However, this configuration will result in decreased performance. If the function can detect the presence of invalid row scenarios, e.g. in cases where sparse_mode is 3 and S_q > S_kv, it will automatically enable row invalidity computation.

- **block_size** (int, *optional*) −Maximum number of tokens per block in the KV cache block for PageAttention. Default: 0. Invalid when Q_S > 1.
- **antiquant_mode** (int, *optional*) −Fake-quantization mode, 0: per-channel (per-channel includes per-tensor), 1: per-token. The per-channel and per-tensor modes can be distinguished by the dimension of the input shape. When the dimension is 1, it runs in per-tensor mode; otherwise, it runs in per-channel mode. Default: 0. Invalid when Q_S > 1.
- **key_antiquant_mode** (int, *optional*) −Fake-quantization mode for the key. 0: per-channel (per-channel includes per-tensor), 1: per-token. Default: 0. Invalid when Q_S > 1.
- **value_antiquant_mode** (int, *optional*) −Fake-quantization mode for the value. 0: per-channel (per-channel includes per-tensor), 1: per-token. Default: 0. Invalid when Q_S > 1.
- **softmax_lse_flag** (bool, *optional*) −Whether to output softmax_lse. Default: False.

Returns

attention_out (Tensor), the attention score with data type of float16, bfloat16 or int8. When the input_layout is BNSD_BSND, the shape is (B, S, N, D). In all other cases, the shape is consistent with the input query shape.

softmax_lse (Tensor), the softmax_lse with data type of float32, obtained by taking the lse (log, sum and exp) of the result of query*key. Specifically, the Ring Attention algorithm first takes the max of the result of query*key, obtaining softmax_max. The result of query*key is then subtracted by softmax_max, followed by taking exp, and then the sum is computed to obtain softmax_sum. Finally, the log of softmax_sum is taken, and softmax_max is added to obtain softmax_lse. The softmax_lse is only calculated when softmax_lse_flag is True, and the shape would be $(B, N, Q\_S, 1)$. If softmax_lse_flag is False, then a tensor with shape (1) filled with zeros would be returned. In graph mode with JitConfig set to O2, please ensure that the softmax_lse_flag is enabled before using softmax_lse; otherwise, an exception will occur.

Constraints:

- Full Inference Scenario (Q_S > 1):
  - Query, key, and value inputs functional usage restrictions:
    - The B axis supports values less than or equal to 65535. If the input type includes int8, or if the input type is float16 or bfloat16 and the D axis is not 16-aligned, the B axis is only supported up to 128.
    - The N axis supports values less than or equal to 256, and the D axis supports values less than or equal to 512.
    - The S axis supports values less than or equal to 20,971,520 (20M). In some long sequence scenarios, if the computation load is too large, it may cause a timeout in the PFA operator (AICore error type with errorStr: "timeout or trap error"). In this case, it is recommended to perform an S split. Note: The computational load is affected by B, S, N, D, etc.; the larger the values, the greater the computational load. Typical long sequence timeout scenarios (where the product of B, S, N, and D is large) include, but are not limited to:
      1. B=1, Q_N=20, Q_S=2097152, D=256, KV_N=1, KV_S=2097152;
      2. B=1, Q_N=2, Q_S=20971520, D=256, KV_N=2, KV_S=20971520;
      3. B=20, Q_N=1, Q_S=2097152, D=256, KV_N=1, KV_S=2097152;
      4. B=1, Q_N=10, Q_S=2097152, D=512, KV_N=1, KV_S=2097152.
    - When the query, key, value, or attention_out type includes int8, the D axis must be 32-aligned. If all types are float16 or bfloat16, the D axis must be 16-aligned.
  - The sparse_mode parameter currently only supports values 0, 1, 2, 3, and 4. Using any other values will result in an error.
    - When sparse_mode = 0, if the atten_mask is None, or if the atten_mask is provided in the left padding scenario, the input parameters pre_tokens and next_tokens are ignored.

    - When sparse_mode = 2, 3, or 4, the shape of the atten_mask must be S,S or 1,S,S or 1,1,S,S, where S must be fixed at 2048, and the user must ensure the atten_mask is a lower triangular matrix. If no atten_mask is provided or if the shape is incorrect, an error will occur.
    - In sparse_mode = 1, 2, 3 scenarios, the pre_tokens and next_tokens inputs are ignored and assigned according to the relevant rules.
  - The KV cache de-quantization only supports queries of type float16, where int8 keys and values are de-quantized to float16. The data range of the input key/value and the antiquant_scale must have a product within the range of (-1, 1). High-performance mode can guarantee precision; otherwise, high-precision mode should be enabled to ensure accuracy.
  - Query left padding scenario:
    - In the query left padding scenario, the formula for calculating the starting point of the query transport is: Q_S - query_padding_size - actual_seq_lengths. The formula for the ending point of the query transport is: Q_S - query_padding_size. The query transport starting point must not be less than 0, and the ending point must not exceed Q_S; otherwise, the results will be incorrect.
    - If the kv_padding_size in the query left padding scenario is less than 0, it will be set to 0.
    - The query left padding scenario must be enabled together with the actual_seq_lengths parameter, otherwise, the default is the query right padding scenario.
    - The query left padding scenario does not support PageAttention and cannot be enabled together with the block_table parameter.
  - KV left padding scenario:
    - In the KV left padding scenario, the formula for calculating the starting point of the key and value transport is: KV_S - kv_padding_size - actual_seq_lengths_kv. The formula for the ending point of the key and value transport is: KV_S - kv_padding_size. The key and value transport starting point must not be less than 0, and the ending point must not exceed KV_S; otherwise, the results will be incorrect.
    - If the kv_padding_size in the KV left padding scenario is less than 0, it will be set to 0.
    - The KV left padding scenario must be enabled together with the actual_seq_lengths_kv parameter, otherwise, the default is the KV right padding scenario.
    - The KV left padding scenario does not support PageAttention and cannot be enabled together with the block_table parameter.
  - pse_shift functional usage restrictions:
    - This function is supported when the query data type is float16, bfloat16, or int8.
    - If the query data type is float16 and pse_shift is enabled, it will force high-precision mode, inheriting the limitations of high-precision mode.
    - Q_S must be greater than or equal to the length of the query S, and KV_S must be greater than or equal to the length of the key S.
  - KV fake quantization parameter separation is not currently supported.
- Incremental Inference Scenario (Q_S is 1):
  - Query, key, and value inputs functional usage restrictions:
    - The B axis supports values less than or equal to 65,536.
    - The N axis supports values less than or equal to 256.
    - The D axis supports values less than or equal to 512.
    - Scenarios where the input types of query, key, and value are all int8 are not supported.

- Page attention scenario:
  - The necessary condition to enable page attention is that the block_table exists and is valid. The key and value are arranged in contiguous memory according to the indices in the block_table. The key and value dtypes supported are float16, bfloat16, and int8. In this scenario, the input_layout parameter for key and value is invalid.
  - block_size is a user-defined parameter, and its value will affect the performance of page attention. When enabling page attention, a non-zero value for block_size must be provided, and the maximum value for block_size is 512.
  - If the input types of key and value are float16 or bfloat16, they must be 16-aligned. If the input types are int8, they must be 32-aligned, with 128 being recommended. In general, page attention can increase throughput but may lead to a performance decrease.
  - In the page attention enabled scenario, when the KV cache layout is (blocknum, block_size, H) and num_key_value_heads * D exceeds 64K, an error will be reported due to hardware instruction constraints. This can be resolved by enabling GQA (reducing num_key_value_heads) or adjusting the KV cache layout to (blocknum, num_key_value_heads, block_size, D).
  - The product of all dimensions of the shape of the key and value tensors in the page attention scenario must not exceed the representable range of int32.
- In the page attention enabled scenario, the input S must be greater than or equal to max_block_num_per_seq * block_size.
  - Enabling attention mask (e.g., mask shape = (B, 1, 1, S))
  - Enabling pse_shift (e.g., pse_shift shape = (B, N, 1, S))
  - Enabling fake quantization in per-token mode (e.g., antiquant_scale and antiquant_offset shapes = (2, B, S)) are also supported.
- KV left padding scenario:
  - In the KV left padding scenario, the formula for calculating the starting point of the KV cache transport is: KV_S - kv_padding_size - actual_seq_lengths. The formula for the endpoint of the KV cache transport is: KV_S - kv_padding_size. If the starting point or endpoint of the KV cache is less than 0, the returned data result will be all zeros.
  - If kv_padding_size is less than 0 in the KV left padding scenario, it will be set to 0.
  - The KV left padding scenario must be enabled together with the actual_seq_lengths parameter, otherwise, it defaults to the KV right padding scenario.
  - The KV left padding scenario must be enabled together with the atten_mask parameter, and the atten_mask must be correctly applied to hide invalid data. Otherwise, accuracy issues may arise.
- pse_shift functional usage restrictions:
  - The data type of pse_shift must match the data type of the query.
  - Only the D axis alignment is supported, meaning the D axis must be divisible by 16.
- KV fake quantization parameter separation:
  - key_antiquant_mode and value_antiquant_mode must be consistent.
  - key_antiquant_scale and value_antiquant_scale must either both be empty or both non-empty.
  - key_antiquant_offset and value_antiquant_offset must either both be empty or both non-empty.
  - When both key_antiquant_scale and value_antiquant_scale are non-empty, their shapes must be consistent.
  - When both key_antiquant_offset and value_antiquant_offset are non-empty, their shapes must be consistent.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> import numpy as np
>>> B, N, S, D = 1, 8, 1024, 128
>>> query = Tensor(np.random.rand(B, N, S, D).astype(np.float16))
>>> key = Tensor(np.random.rand(B, N, S, D).astype(np.float16))
>>> value = Tensor(np.random.rand(B, N, S, D).astype(np.float16))
>>> out = ops.fused_infer_attention_score(query, key, value, num_heads=N, input_layout='BNSD
↪')
>>> print(out[0].shape)
(1, 8, 1024, 128)
```

mindspore.ops.speed_fusion_attention

mindspore.ops.**speed_fusion_attention**(*query*, *key*, *value*, *head_num*, *input_layout*, ***, *pse=None*, *padding_mask=None*, *atten_mask=None*, *scale=1.0*, *keep_prob=1.0*, *pre_tokens=2147483647*, *next_tokens=2147483647*, *inner_precise=0*, *prefix=None*, *actual_seq_qlen=None*, *actual_seq_kvlen=None*, *sparse_mode=0*, *gen_mask_parallel=True*, *sync=False*, *pse_type=1*, *q_start_idx=None*, *kv_start_idx=None*)

The interface is used for self-attention fusion computing. If *pse_type* is `1` , calculation formula is:

$$attention\_out = Dropout(Softmax(Mask(scale * (pse + query * key^T), atten\_mask)), keep\_prob) * value$$

If *pse_type* is other valid value, calculation formula is:

$$attention\_out = Dropout(Softmax(Mask(scale * (query * key^T) + pse, atten\_mask)), keep\_prob) * value$$

- B: Batch size. Value range 1 to 2k.
- S1: Sequence length of query. Value range 1 to 512k.
- S2: Sequence length of key and value. Value range 1 to 512k.
- N1: Num heads of query. Value range 1 to 256.
- N2: Num heads of key and value, and N2 must be a factor of N1.
- D: Head size. The value ranges is a multiple of 16, with the max value of 512.
- H1: Hidden size of query, which equals to N1 * D.
- H2: Hidden size of key and value, which equals to N2 * D.

Warning:

- This is an experimental API that is subject to change or deletion.
- Only support on Atlas A2 training series.

Note: This interface is not supported in graph mode (mode=mindspore.GRAPH_MODE).

Parameters

- **query** (Tensor) −The query tensor. Input tensor of shape $(B, S1, H1)$, $(B, N1, S1, D)$, $(S1, B, H1)$, $(B, S1, N1, D)$ or $(T1, N1, D)$.
- **key** (Tensor) −The key tensor. Input tensor of shape $(B, S2, H2)$, $(B, N2, S2, D)$, $(S2, B, H2)$, $(B, S2, N2, D)$ or $(T2, N2, D)$.
- **value** (Tensor) −The value tensor. Input tensor of shape $(B, S2, H2)$, $(B, N2, S2, D)$, $(S2, B, H2)$, $(B, S2, N2, D)$ or $(T2, N2, D)$. The *key* and *value* should have the same shape.
- **head_num** (*int*) −The head num of query, equal to N1.
- **input_layout** (*str*) −Specifies the layout of input *query*, *key* and *value*. The value can be "BSH" , "BNSD" , "SBH" , "BSND" or "TND" . "TND" is an experimental format. When *input_layout* is "TND" , the following restrictions must be met. There are two lists that represent the length of the input sequence: list_seq_q and list_seq_k. Each value in the list indicates the length of the sequence in the batch. For example, list_seq_q = [4, 2, 6], list_seq_k = [10, 3, 9]. The element of list indicate S. T1 is sum(list_seq_q) = 12, T2 is sum(list_seq_k) = 22. max_seqlen_q = max(list_seq_q), max_seqlen_k = max(list_seq_k). qk_pointer = sum(list_seq_q * list_seq_k), which is the sum of the element multiplication.
  - The lengths of two lists are the same, and size of list is batch. batch is less than or equal to 1024.
  - When *input_layout* is "TND" , *actual_seq_qlen* and *actual_seq_kvlen* must be not None . Otherwise, they are None .
  - The *actual_seq_qlen* and *actual_seq_kvlen* are the cumulative sum of sequence of key/value, so they must be non-decreasing.
  - If *pse* is not None , list_seq_q and list_seq_k must be same. The maximum value of list_seq_q and list_seq_k is greater than 1024. *pse* should be $(B, N1, 1024, S2)$ and $(1, N1, 1024, S2)$, and S2 is equal to max_seqlen_k.
  - *atten_mask* must be a lower trianglar matrix, so *sparse_mode* should be 2 or 3. The shape of *atten_mask* should be $(2048, 2048)$.
  - Prefix is None .
  - *next_tokens* is 0, and *pre_tokens* is not less than max_seqlen_q.
  - When *sparse_mode* is 3, S1 of each batch should be less than or equal to S2.
  - 0 should not exist in list_seq_k.

Keyword Arguments

- **pse** (Tensor, *optional*) −The position embedding code, dtype is same as *query*. Default: None . If S is greater than 1024 and the mask of the lower triangle is used, enter only the inverse 1024 lines of the lower triangle for memory optimization. Input tensor of shape $(B, N1, S1, S2)$, $(1, N1, S1, S2)$, $(B, N1, 1024, S2)$, $(1, N1, 1024, S2)$.
  - ALiBi scenario: *pse* must meet the ALiBi rule, and *sparse_mode* is 2 or 3 for the lower triangle. In this scenario, *pse* is $(B, N1, 1024, S2)$, $(1, N1, 1024, S2)$.
  - Non-ALiBi scenario: *pse* is $(B, N1, S1, S2)$, $(1, N1, S1, S2)$.
  - The shape of *pse* should be $(B, N1, 1024, S2)$ and $(1, N1, 1024, S2)$ when *input_layout* is "TND" .
  - If *pse_type* is 2 or 3, dtype of *pse* must be float32, and shape of *pse* should be $(B, N1)$ or $(N1,)$.
- **padding_mask** (Tensor, *optional*) −Reserved parameter. Not implemented yet. Default: None .
- **atten_mask** (Tensor, *optional*) −The attention mask tensor. For each element, 0/False indicates retention and 1/True indicates discard. Input tensor of shape $(B, N1, S1, S2)$, $(B, 1, S1, S2)$, $(S1, S2)$ or $(2048, 2048)$. Default: None .
  - In compression scenario, *sparse_mode* is 2, 3, or 4, *atten_mask* must be $(2048, 2048)$.

  - When *sparse_mode* is 5, *atten_mask* must be $(B, N1, S1, S2)$, $(B, 1, S1, S2)$.
  - When *sparse_mode* is 0 and 1, *atten_mask* should be $(B, N1, S1, S2)$, $(B, 1, S1, S2)$, $(S1, S2)$.

- **scale** (float, *optional*) –The scale factor of score. Generally, the value is 1.0 / (D ** 0.5). Default: `1.0` .
- **keep_prob** (float, *optional*) –The keep probability of dropout. Value range is (0.0, 1.0]. Default: `1.0` .
- **pre_tokens** (int, *optional*) –Parameter for sparse computation, represents how many tokens are counted forward. When *sparse_mode* is set to 1, 2, 3, or 5, this parameter does not take effect. Default: `2147483647` .
- **next_tokens** (int, *optional*) –Parameter for sparse computation, represents how many tokens are counted backward. When *sparse_mode* is set to 1, 2, 3, or 5, this parameter does not take effect. Default: `2147483647` . The value of pre_tokens corresponds to S1, and the value of next_tokens corresponds to S2. They define the valid area on the *atten_mask* matrix. It must ensure that the band is not empty. The following values are not allowed:
  - pre_tokens < 0 and next_tokens < 0.
  - (pre_tokens < 0 and next_tokens >= 0) and (next_tokens < abs(pre_tokens) or abs(pre_tokens) >= S2).
  - (pre_tokens >= 0 and next_tokens < 0) and (abs(next_tokens) > pre_tokens or abs(next_tokens) >= S1).
- **inner_precise** (int, *optional*) –The parameter is reserved and not implemented yet. Default: `0` .
- **prefix** (*Union[tuple[int], list[int]], optional*) –N value of each Batch in the prefix sparse calculation scenario. Input tensor of shape $(B,)$. B max value 32. Not none only when sparse_mode is 5. If S1 > S2, N ranges from 0 to S2. If S1 <= S2, N ranges from S2 - S1 to S2. Default: `None` .
- **actual_seq_qlen** (*Union[tuple[int], list[int]], optional*) –Size of query corresponding to each batch, array with increasing values and the last value equal to T1. Default: `None` .
- **actual_seq_kvlen** (*Union[tuple[int], list[int]], optional*) –Size of key and value corresponding to each batch, array with increasing values and the last value equal to T2. Default: `None` .
- **sparse_mode** (int, *optional*) –Indicates sparse mode. Default `0` .
  - 0: Indicates the defaultMask mode. If *atten_mask* is not passed, the mask operation is not performed, and preTokens and nextTokens(internally assigned as INT_MAX) are ignored. If passed in, the full *atten_mask* matrix (S1 * S2) needs to be passed in, indicating that the part between preTokens and nextTokens needs to be calculated.
  - 1: Represents allMask, that is, passing in the complete *atten_mask* matrix.
  - 2: Representing the leftUpCausal mode corresponds to the lower triangle scenario divided by the left vertex, and the optimized *atten_mask* matrix (2048*2048) is required.
  - 3: Representing the rightDownCausal model corresponds to the lower triangle scene divided by the lower right vertex, and the optimized *atten_mask* matrix (2048*2048) is required.
  - 4: Represents the band scenario, that is, the part between counting preTokens and nextTokens, and the optimized *atten_mask* matrix (2048*2048) is required.
  - 5: Represents the prefix scenario, that is, on the basis of rightDownCasual, a matrix with length S1 and width N is added to the left side. The value of N is obtained by the new input prefix, and the N value of each Batch axis is different. Currently not enabled.
  - 6: Represents the global scenario. Currently not enabled.
  - 7: Represents the dilated scenario. Currently not enabled.
  - 8: Represents the block_local scenario. Currently not enabled.
- **gen_mask_parallel** (bool, *optional*) –Debug parameter, a switch to control dropout_gen_mask execution method. If `True` , dropout_gen_mask is executed in parallel. If `False` , execution is serial. Not implemented yet. Default: `True` .

- **sync** (bool, *optional*) –Debug parameter, a switch to control dropout_gen_mask execution method. If `True` , dropout_gen_mask is executed synchronously. If `False` , execution is asynchronous. Not implemented yet. Default: `False` .
- **pse_type** (int, *optional*) –Indicates how to use *pse*. Default `1` .
  - 0: *pse* is passed from outside, and the calculation process is to first mul *scale* and then add *pse*.
  - 1: *pse* is passed from outside, and the calculation process is to add *pse* first and then mul *scale*.
  - 2: *pse* is generated internally and generates standard alibi position information. The internally generated alibi matrix 0 line is aligned with the upper left corner of $query * key^T$.
  - 3: *pse* is generated internally, and the generated alibi position information is based on the standard and then the square root of sqrt is done. The internally generated alibi matrix 0 line is aligned with the upper left corner of $query * key^T$.
- **q_start_idx** (*Union[tuple[int], list[int]], optional*) –Int array with length 1. Default: `None` . When pse_type is configured as `2` or `3` , it indicates the number of cells that the internally generated alibi code is offset in the S1 direction. A positive number indicates that 0 moves diagonally upward.
- **kv_start_idx** (*Union[tuple[int], list[int]], optional*) –Int array with length 1. Default: `None` . When pse_type is configured as `2` or `3` , it indicates the number of cells that the internally generated alibi code is offset in the S2 direction. A positive number indicates that 0 moves diagonally upward.

Returns

A tuple of tensors containing *attention_out*, *softmax_max*, *softmax_sum*, *softmax_out*, *seed*, *offset* and *numels* .

- *attention_out* is the output of attention, it's shape, and data type are the same as the query.
- *softmax_max* is the max intermediate result calculated by Softmax, used for grad calculation.
- *softmax_sum* is the sum intermediate result calculated by Softmax, used for grad calculation.
- *softmax_out* is a reserved parameter.
- *seed* is generated seed, used for Dropout.
- *offset* is generated offset, used for Dropout.
- *numels* is the length of generated dropout_mask.

Raises

- **TypeError** –*query*, *key* and *value* don't have the same dtype.
- **TypeError** –Dtype of *atten_mask* is not bool or uint8.
- **TypeError** –*scale* or *keep_prob* is not a float number.
- **TypeError** –*input_layout* is not a string.
- **TypeError** –*head_num* is not an int.
- **TypeError** –*sparse_mode* is not an int.
- **TypeError** –*pse* is not Tensor type.
- **TypeError** –*padding_mask* is not Tensor type.
- **TypeError** –*atten_mask* is not Tensor type.
- **TypeError** –*pse_type* is not an int.
- **ValueError** –*input_layout* is a string but not valid.

- **ValueError** –The specified value of *sparse_mode* is invalid.
- **ValueError** –The specified value of *pse_type* is invalid.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import mindspore.common.dtype as mstype
>>> import numpy as np
>>> from mindspore import ops, Tensor
>>> query = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> key = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> value = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> head_num = 4
>>> input_layout = "BSH"
>>> output = ops.speed_fusion_attention(query, key, value, head_num, input_layout)
>>> print(output[0].shape)
(2, 4, 64)
```

mindspore.ops.group_norm

`mindspore.ops.`**`group_norm`**(*input*, *num_groups*, *weight=None*, *bias=None*, *eps=1e-5*)

Group Normalization over a mini-batch of inputs.

Group Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Group Normalization. Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization, and it performs very stable over a wide range of batch size. γ and β are trainable scale and shift. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

where γ is *weight*, β is *bias*, ϵ is *eps*.

Parameters

- **input** (`Tensor`) –The input feature with shape $(N, C, *)$ where $*$ means, any number of additional dimensions.
- **num_groups** (*int*) –The number of groups to be divided along the channel dimension.
- **weight** (`Tensor, optional`) –The shape $(C,)$, Default: `None`, has the same data type with *input*.
- **bias** (`Tensor, optional`) –The shape $(C,)$, Default: `None`, has the same data type with *input*.
- **eps** (*float, optional*) –A value added to the denominator for numerical stability. Default: `1e-5` .

Returns

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the *input*.

Raises

- **TypeError** –If *num_groups* is not an int.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_groups* is less than 1.

- **ValueError** –If *C* (the second parameter of dimensions of *input*) is not divided by *num_groups*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import ops
>>> x = ms.Tensor(np.ones([1, 2, 4, 4], np.float32))
>>> output = ops.group_norm(x, 2)
>>> print(output)
[[[[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
  [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]]]
```

mindspore.ops.layer_norm

`mindspore.ops.`**`layer_norm`**(*input*, *normalized_shape*, *weight=None*, *bias=None*, *eps=1e-5*)

Applies the Layer Normalization on the mini-batch input.

Layer normalization is widely used in recurrent neural networks. Apply normalization to the mini-batch input of a single training case. LayerNorm is described in the paper Layer Normalization.

Unlike batch normalization, layer normalization performs the exact same calculations at training and test time. Applies to all channels and pixels, even batch_size=1. The formula is as follows:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

where γ is the weight value learned through training, β is the bias value learned through training.

Parameters

- **input** (`Tensor`) –The shape of input is *(N, *)*, where * represents any additional dimension.
- **normalized_shape** (*`Union(int, tuple[int], list[int])`*) –The normalized shape of *input* for LayerNorm. *normalized_shape* equal to *input_shape[begin_norm_axis:]*, where *begin_norm_axis* represents the axis where normalization begins.
- **weight** (`Tensor, optional`) –Learnable parameter γ . Tensor of shape *normalized_shape*. Default: `None`, has the same data type with *input*. Initialized to `1` when *weight* is None.
- **bias** (`Tensor, optional`) –Learnable parameter β . Tensor of shape *normalized_shape*. Default: `None`, has the same data type with *input*. Initialized to `0` when *bias* is None.
- **eps** (*`float, optional`*) –A value added to the denominator for numerical stability(ϵ). Default: `1e-5` .

Returns

Tensor. The normalized tensor, has the same type and shape as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *normalized_shape* is not an integer, a list or a tuple.
- **TypeError** –If *eps* is not a float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[1, 2, 3], [1, 2, 3]]), mindspore.float32)
>>> normalized_shape = (3,)
>>> gamma = Tensor(np.ones(normalized_shape), mindspore.float32)
>>> beta = Tensor(np.zeros(normalized_shape), mindspore.float32)
>>> eps = 1e-7
>>> output = ops.layer_norm(input_x, normalized_shape, gamma, beta, eps)
>>> print(output)
[[-1.2247448 0. 1.2247448]
 [-1.2247448 0. 1.2247448]]
```

mindspore.ops.lp_pool1d

mindspore.ops.**lp_pool1d**(*x*, *norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applying 1D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.

Typically the input is of shape (N, C, L_{in}) or (C, L_{in}), the output is of shape (N, C, L_{out}) or (C, L_{out}).

$$L_{out} = \left\lfloor \frac{L_{in} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

The operation is as follows.

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **x** (Tensor) –Tensor of shape (N, C, L_{in}) or (C, L_{in}).
- **norm_type** (*Union[int, float]*) –Type of normalization, represents p in the formula, can not be 0,
  - if p = 1, the result obtained is the sum of elements in the pool nucleus(Proportional to average pooling).
  - if p = ∞, the result is the result of maximum pooling.
- **kernel_size** (*int*) –The size of kernel window.

- **stride** (*int*) −The distance of kernel moving, an int number that represents the width of movement is stride. Default: `None` , which indicates the moving step is *kernel_size* .
- **ceil_mode** (*bool*) −Whether to use ceil or floor to calculate output shape. Default: `False` .

Returns

- **output** (Tensor) - LPPool1d result, with shape (N, C, L_{out}) or (C, L_{out}), It has the same data type as x, where

$$L_{out} = \left\lfloor \frac{L_{in} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

Raises

- **TypeError** −If x is not a Tensor.
- **TypeError** −If *kernel_size* or *stride* is not an int.
- **TypeError** −If *ceil_mode* is not a bool.
- **TypeError** −If *norm_type* is neither float nor int.
- **ValueError** −If *norm_type* is equal to 0.
- **ValueError** −If *kernel_size* or *stride* is less than 1.
- **ValueError** −If length of shape of x is not equal to 2 or 3.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.arange(2 * 3 * 4).reshape((2, 3, 4)), dtype=ms.float32)
>>> out = ops.lp_pool1d(x, norm_type=1, kernel_size=3, stride=1, ceil_mode=False)
>>> print(out)
[[[ 3.  6.]
  [15. 18.]
  [27. 30.]]
 [[39. 42.]
  [51. 54.]
  [63. 66.]]]
```

mindspore.ops.lp_pool2d

`mindspore.ops.`**lp_pool2d**(*x*, *norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applying 2D LPPooling operation on an input Tensor can be regarded as forming a 2D input plane.

Typically the input is of shape (N, C, H_{in}, W_{in}), the output is of shape (N, C, H_{in}, W_{in}), with the same shape as input, the operation is as follows.

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **x** (Tensor) −Tensor of shape (N, C, H_{in}, W_{in}).
- **norm_type** (*Union[int, float]*) −Type of normalization, represents p in the formula, can not be 0,
  - if p = 1, the result obtained is the sum of elements in the pool nucleus(Proportional to average pooling).
  - if p = ∞, the result is the result of maximum pooling.
- **kernel_size** (*Union[int, tuple[int]]*) −The size of kernel window. The data type of kernel_size must be int and the value represents the height and width, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]]*) −The distance of kernel moving, an int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: None , which indicates the moving step is *kernel_size* .
- **ceil_mode** (*bool*) −Whether to use ceil or floor to calculate output shape. Default: False .

Returns

- **output** (Tensor) - LPPool2d result, with shape (N, C, H_{in}, W_{in}), It has the same data type as *x*, where

$$H_{out} = \left\lfloor \frac{H_{in} - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

Raises

- **TypeError** −If *x* is not a Tensor.
- **TypeError** −If *kernel_size* or *stride* is neither int nor tuple.
- **TypeError** −If *ceil_mode* is not a bool.
- **TypeError** −If *norm_type* is neither float nor int.
- **ValueError** −If *norm_type* is equal to 0.
- **ValueError** −If *kernel_size* or *stride* is less than 1.
- **ValueError** −If *kernel_size* or *stride* is a tuple whose length is not equal to *2*.
- **ValueError** −If length of shape of *x* is not equal to 4.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> import numpy as np
>>> x = Tensor(np.arange(2 * 3 * 4 * 5).reshape((2, 3, 4, 5)), dtype=ms.float32)
>>> out = ops.lp_pool2d(x, norm_type=1, kernel_size=3, stride=1, ceil_mode=False)
>>> print(out)
[[[[  54.   63.   72.]
   [  99.  108.  117.]]
  [[ 234.  243.  252.]
   [ 279.  288.  297.]]
  [[ 414.  423.  432.]
   [ 459.  468.  477.]]]
 [[[ 594.  603.  612.]
   [ 639.  648.  657.]]
  [[ 774.  783.  792.]
   [ 819.  828.  837.]]
  [[ 954.  963.  972.]
   [ 999. 1008. 1017.]]]]
```

mindspore.ops.lrn

mindspore.ops.**lrn**(*x*, *depth_radius=5*, *bias=1.0*, *alpha=1.0*, *beta=0.5*, *norm_region='ACROSS_CHANNELS'*)

Local Response Normalization.

> **Warning:** lrn is deprecated on Ascend due to potential accuracy problem. It's recommended to use other normalization methods, e.g. *mindspore.ops.batch_norm*.

$$b_c = a_c \left(k + \frac{\alpha}{n} \sum_{c'=\max(0,c-n/2)}^{\min(N-1,c+n/2)} a_{c'}^2 \right)^{-\beta}$$

where the a_c indicates the specific value of the pixel corresponding to c in feature map; where the $n/2$ indicates the *depth_radius*; where the k indicates the *bias*; where the α indicates the *alpha*; where the β indicates the *beta*.

Parameters

- **depth_radius** (*int*) –Half-width of the 1-D normalization window with the shape of 0-D. Default: `5` .
- **bias** (*float*) –An offset (usually positive to avoid dividing by 0). Default: `1.0` .
- **alpha** (*float*) –A scale factor, usually positive. Default: `1.0` .
- **beta** (*float*) –An exponent. Default: `0.5` .
- **norm_region** (*str*) –Specifies normalization region. Options: `"ACROSS_CHANNELS"` . Default: `"ACROSS_CHANNELS"` .
- **x** (Tensor) –A 4-D Tensor with float16 or float32 data type.

Returns

Tensor, with the same shape and data type as *x*.

Raises

- **TypeError** –If *depth_radius* is not an int.
- **TypeError** –If *bias*, *alpha* or *beta* is not a float.
- **TypeError** –If *norm_region* is not a str.
- **TypeError** –If *x* is not a Tensor.

Supported Platforms:

`GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[[[0.1], [0.2]],
...                            [[0.3], [0.4]]]]), mindspore.float32)
>>> output = ops.lrn(input_x)
>>> print(output)
[[[[0.09534626]
   [0.1825742 ]]
  [[0.2860388 ]
   [0.3651484 ]]]]
```

mindspore.ops.max_pool2d

mindspore.ops.**max_pool2d**(*x*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

Performs a 2D max pooling on the input Tensor.

Typically, the input is a Tensor with shape $(N_{in}, C_{in}, H_{in}, W_{in})$, outputs regional maximum in the (H_{in}, W_{in})-dimension. Given *kernel_size* $ks = (h_{ker}, w_{ker})$ and *stride* $s = (s_0, s_1)$, the operation is as follows:

$$\text{output}(N_i, C_j, h, w) = \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times h + m, s_1 \times w + n)$$

Parameters

- **x** (Tensor) –Tensor of shape $(N_{in}, C_{in}, H_{in}, W_{in})$ with data type of int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32 or float64 in CPU or GPU while that of uint16 in Ascend.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value and arg value, is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: `None` , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]]*) –An int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: `0` .
- **dilation** (*Union[int, tuple[int]]*) –Control the stride of elements in the kernel. Default: `1` .
- **return_indices** (*bool*) –Whether to output the indices of max value. Default: `False` .
- **ceil_mode** (*bool*) –Whether to use ceil instead of floor to calculate output shape. Default: `False` .

Returns

If *return_indices* is `False` , return a Tensor *output*, else return a tuple (*output*, *argmax*).

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, H_{out}, W_{out})$. It has the same data type as x.

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

- **argmax** (Tensor) - Index corresponding to the maximum value. In CPU and GPU, data type is int64 while that is uint16 in Ascend. It will be return only when *return_indices* is True.

Raises

- **TypeError** – If x is not a Tensor.
- **ValueError** – If length of shape of x is not equal to 4.
- **TypeError** – If *kernel_size* , *stride* , *padding* or *dilation* is not int or tuple.
- **ValueError** – If *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *padding* is less than 0.
- **ValueError** – If *padding* is more than half of *kernel_size*.
- **TypeError** – If *ceil_mode* is not bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(20 * 16 * 50 * 32).reshape((20, 16, 50, 32)), mindspore.float32)
>>> output_tensor, argmax = ops.max_pool2d(x, kernel_size=(3, 2), stride=(2, 1), return_
↪indices=True)
>>> print(output_tensor.shape)
(20, 16, 24, 31)
>>> print(argmax.shape)
(20, 16, 24, 31)
```

mindspore.ops.max_pool3d

`mindspore.ops.`**`max_pool3d`**(*x*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *ceil_mode=False*, *return_indices=False*)

Performs a 3D max pooling on the input Tensor.

Typically the input is a Tensor with shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$, outputs regional maximum in the (D_{in}, H_{in}, W_{in})-dimension. Given *kernel_size* $ks = (d_{ker}, h_{ker}, w_{ker})$ and *stride* $s = (s_0, s_1, s_2)$, the operation is as follows:

$$\text{output}(N_i, C_j, d, h, w) = \max_{l=0,\ldots,d_{ker}-1} \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Parameters

- **x** (Tensor) −Tensor of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$ with data type of int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32 or float64.
- **kernel_size** (*Union[int, tuple[int]]*) −The size of kernel used to take the maximum value and arg value, is an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively.
- **stride** (*Union[int, tuple[int]]*) −The distance of kernel moving, an int number that represents the depth, height and width of movement are both stride, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: None , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]]*) −An int number that represents the depth, height and width of movement are both strides, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: 0 .
- **dilation** (*Union[int, tuple[int]]*) −Control the stride of elements in the kernel. Default: 1 .
- **ceil_mode** (*bool*) −Whether to use ceil instead of floor to calculate output shape. Default: False .
- **return_indices** (*bool*) −Whether to output the indices of max value. Default: False .

Returns

If *return_indices* is False, return a Tensor *output*, else return a tuple (*output*, *argmax*).

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, D_{out}, H_{out}, W_{out})$. It has the same data type as *x*.

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

- **argmax** (Tensor) - Index corresponding to the maximum value. Data type is int64. It will be returned only when *return_indices* is True .

Raises

- **TypeError** −If *x* is not a Tensor.
- **ValueError** −If length of shape of *x* is not equal to 5.
- **TypeError** −If *kernel_size* , *stride* , *padding* or *dilation* is not int or tuple.
- **ValueError** −If *kernel_size* or *stride* is less than 1.
- **ValueError** −If *padding* is less than 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(2 * 1 * 2 * 2 * 2).reshape((2, 1, 2, 2, 2)), mindspore.float32)
>>> output_tensor, argmax = ops.max_pool3d(x, kernel_size=2, stride=1, padding=1, return_
↪indices=True)
>>> print(output_tensor.shape)
(2, 1, 3, 3, 3)
>>> print(argmax.shape)
(2, 1, 3, 3, 3)
```

mindspore.ops.max_unpool1d

mindspore.ops.**max_unpool1d**(*x*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

Computes the inverse of *max_pool1d*.

max_unpool1d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}) or (C, H_{in}), and the output is of shape (N, C, H_{out}) or (C, H_{out}). The operation is as follows.

$$H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$

Parameters

- **x** (Tensor) –The input Tensor to invert. Tensor of shape (N, C, H_{in}) or (C, H_{in}).
- **indices** (Tensor) –Index of maximum value. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, H_{in} - 1]$. Data type must be in int32 or int64.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value.
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, If stride is 0, (0) or None , then stride equal to kernel_size. Default: None , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]]*) –The pad value to be filled. Default: 0 .
- **output_size** (*tuple[int], optional*) –The output shape. Default: None . If output_size == (), then the shape of output computed by *kernel_size*, *stride* and *padding*. If output_size != (), then output_size must be (N, C, H) , (C, H) or (H) and output_size must belong to $[(N, C, H_{out} - stride[0]), (N, C, H_{out} + stride[0])]$.

Returns

Tensor, with shape (N, C, H_{out}) or (C, H_{out}), with the same data type with *x*.

Raises

- **TypeError** –If data type of *x* or *indices* is not supported.
- **TypeError** –If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** –If numbers in *stride*, *padding* (also support 0 and (0)) or *kernel_size* is not positive.
- **ValueError** –If the shapes of *x* and *indices* are not equal.
- **ValueError** –If *x* whose length is not 2 or 3.
- **ValueError** –If type of *output_size* is not tuple.
- **ValueError** –If *output_size* whose length is not 0, 2 or 3.
- **ValueError** –If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[2, 4, 6, 8]]).astype(np.float32))
>>> indices = Tensor(np.array([[1, 3, 5, 7]]).astype(np.int64))
>>> output = ops.max_unpool1d(x, indices, kernel_size =2, stride=2, padding=0)
>>> print(output.asnumpy())
[[0. 2. 0. 4. 0. 6. 0. 8.]]
```

mindspore.ops.max_unpool2d

mindspore.ops.**max_unpool2d**(*x*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

Computes the inverse of *max_pool2d*.

max_unpool2d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}), and the output is of shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}). The operation is as follows.

$$H_{out} = (Hin - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$
$$W_{out} = (Win - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1]$$

Parameters

- **x** (Tensor) – The input Tensor to invert. Tensor of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- **indices** (Tensor) – Max values' index represented by the indices. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]]*) – The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: `None` , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]]*) – The pad value to be filled. Default: `0` . If *padding* is an integer, the paddings of height and width are the same, equal to padding. If *padding* is a tuple of two integers, the padding of height and width equal to padding[0] and padding[1] correspondingly.
- **output_size** (*tuple[int], optional*) – The target output size. Default: `None` . If output_size == (), then the shape of output computed by *kernel_size*, *stride* and *padding*. If output_size != (), then output_size must be (N, C, H, W) , (C, H, W) or (H, W) and output_size must belong to $[(N, C, H_{out} - stride[0], W_{out} - stride[1]), (N, C, H_{out} + stride[0], W_{out} + stride[1])]$.

Returns

Tensor, with shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), with the same data type with *x*.

Raises

- **TypeError** – If data type of *x* or *indices* is not supported.
- **TypeError** – If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** – If numbers in *stride*, *padding* (also support 0 and (0, 0)) or *kernel_size* is not positive.

- **ValueError** –If the shape of *x* and *indices* are not equal.
- **ValueError** –If *kernel_size*, *stride* or *padding* is a tuple whose length is not equal to 2.
- **ValueError** –If *x* whose length is not 3 or 4.
- **ValueError** –If *output_size* whose type is not tuple.
- **ValueError** –If *output_size* whose length is not 0, 3 or 4.
- **ValueError** –If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[[0, 1], [8, 9]]]]).astype(np.float32))
>>> indices = Tensor(np.array([[[[0, 1], [2, 3]]]]).astype(np.int64))
>>> output = ops.max_unpool2d(x, indices, kernel_size=1, stride=1, padding=0)
>>> print(output.asnumpy())
[[[[0. 1.]
   [8. 9.]]]]
```

mindspore.ops.max_unpool3d

mindspore.ops.**max_unpool3d**(*x*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

Computes the inverse of *mindspore.ops.max_pool3d()*.

max_unpool3d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$, and the output is of shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$. The operation is as follows.

$$
\begin{aligned}
D_{out} &= (Din - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0] \\
H_{out} &= (Hin - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1] \\
W_{out} &= (Win - 1) \times stride[2] - 2 \times padding[2] + kernel\_size[2]
\end{aligned}
$$

Parameters

- **x** (Tensor) –The input Tensor to invert. Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$.
- **indices** (Tensor) –Max values' index represented by the indices. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, D_{in} \times H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value, an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively.
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number that represents the depth, height and width of movement are both stride, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: `None` , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]]*) –The pad value to be filled. Default: `0` . If *padding* is an integer, the paddings of depth, height and width are the same, equal to padding. If *padding* is a tuple of three integers, the padding of depth, height and width equal to padding[0], padding[1] and padding[2] correspondingly.

- **output_size** (*tuple[int], optional*) – The output size. Default: `None`. If output_size == (), then the shape of output computed by *kernel_size*, *stride* and *padding*. If output_size != (), then output_size must be (N, C, D, H, W) or (C, D, H, W) or (D, H, W) and output_size must belong to $[(N, C, D_{out} - stride[0], H_{out} - stride[1], W_{out} - stride[2]), (N, C, D_{out} + stride[0], H_{out} + stride[1], W_{out} + stride[2])]$.

Returns

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$, with the same data type with *x*.

Raises

- **TypeError** – If data type of *x* or *indices* is not supported.
- **TypeError** – If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** – If numbers in *stride* or *padding* (also support 0 and (0, 0, 0)) or *kernel_size* is not positive.
- **ValueError** – If the shape of *x* and *indices* are not equal.
- **ValueError** – If *kernel_size*, *stride* or *padding* is a tuple whose length is not equal to 3.
- **ValueError** – If *x* whose length is not 4 or 5.
- **ValueError** – If *output_size* whose length is not 0, 4 or 5.
- **ValueError** – If *output_size* whose type is not tuple.
- **ValueError** – If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[[[0, 1], [8, 9]]]]]).astype(np.float32))
>>> indices= Tensor(np.array([[[[[0, 1], [2, 3]]]]]).astype(np.int64))
>>> output = ops.max_unpool3d(x, indices, kernel_size=2, stride=1, padding=0)
>>> print(output)
[[[[[0. 1. 8.]
    [9. 0. 0.]
    [0. 0. 0.]]
   [[0. 0. 0.]
    [0. 0. 0.]
    [0. 0. 0.]]]]]
```

mindspore.ops.moe_token_permute

mindspore.ops.**moe_token_permute**(*tokens*, *indices*, *num_out_tokens=None*, *padded_mode=False*)

Permute the *tokens* based on the *indices*. Token with the same index will be grouped together.

Warning:

- It is only supported on Atlas A2 Training Series Products.
- The input *tokens* only supports the bfloat16 data type in the current version.

- When *indices* is 2-D, the size of the second dim must be less than or equal to 512.

Parameters

- **tokens** (Tensor) −The input token tensor to be permuted. The dtype is bfloat16. The shape is $(num\_tokens, hidden\_size)$, where *num_tokens* and *hidden_size* are positive integers.
- **indices** (Tensor) −The tensor specifies indices used to permute the tokens. The dtype is int32 or int64. The shape is $(num\_tokens, topk)$ or $(num\_tokens,)$, where *num_tokens* and *topk* are positive integers. If the shape is the latter case, *topk* is implied to be 1.
- **num_out_tokens** (*int, optional*) −The effective output token count, when enabling the capacity factor, should equal the number of tokens not dropped. It should be non-negative integer. Default: None, meaning no tokens are dropped.
- **padded_mode** (*bool, optional*) −If True, indicating the indices are padded to denote selected tokens per expert. It can only be False currently. Default: False .

Returns

tuple (Tensor), tuple of 2 tensors, containing the permuted tokens and sorted indices.

- **permuted_tokens** (Tensor) - The permuted tensor of the same dtype as *tokens*.
- **sorted_indices** (Tensor) - The indices Tensor of dtype int32, corresponds to permuted tensor.

Raises

- **TypeError** −If *tokens* or *indices* is not a Tensor.
- **TypeError** −If dtype of *tokens* is not bfloat16.
- **TypeError** −If dtype of *indices* is not int32 or int64.
- **TypeError** −If specified *num_out_tokens* is not an integer.
- **TypeError** −If specified *padded_mode* is not a bool.
- **ValueError** −If second dim of *indices* is greater than 512 when exists.
- **ValueError** −If *padded_node* is set to True.
- **ValueError** −If *tokens* is not 2-D or *indices* is not 1-D or 2-D Tensor.
- **RuntimeError** −If first dimensions of *tokens* and *indices* are not consistent.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> tokens = Tensor([[1, 1, 1],
...                  [7, 7, 7],
...                  [2, 2, 2],
...                  [1, 1, 1],
...                  [2, 2, 2],
...                  [3, 3, 3]], dtype=mindspore.bfloat16)
>>> indices = Tensor([5, 0, 3, 1, 2, 4], dtype=mindspore.int32)
>>> out = ops.moe_token_permute(tokens, indices)
>>> print(out)
(Tensor(shape=[6, 3], dtype=BFloat16, value=
[[7, 7, 7],
 [1, 1, 1],
 [2, 2, 2],
 [2, 2, 2],
 [3, 3, 3],
 [1, 1, 1]]), Tensor(shape=[6], dtype=Int32, value= [5, 0, 3, 1, 2, 4]))
```

mindspore.ops.moe_token_unpermute

mindspore.ops.**moe_token_unpermute**(*permuted_tokens*, *sorted_indices*, *probs=None*, *padded_mode=False*, *restore_shape=None*)

Unpermute a tensor of permuted tokens based on sorted indices, and optionally merge the tokens with their corresponding probabilities.

Warning:

- It is only supported on Atlas A2 Training Series Products.
- The inputs *permuted_tokens* and *probs* only support the bfloat16 data type in the current version.
- *sorted_indices* must not have duplicate values, otherwise the result is undefined.

Parameters

- **permuted_tokens** (Tensor) –The tensor of permuted tokens to be unpermuted. The shape is $[num\_tokens * topk, hidden\_size]$, where *num_tokens*, *topk* and *hidden_size* are positive integers.
- **sorted_indices** (Tensor) –The tensor of sorted indices used to unpermute the tokens. The shape is $[num\_tokens * topk,]$, where *num_tokens* and *topk* are positive integers. It only supports the int32 data type.
- **probs** (Tensor, *optional*) –The tensor of probabilities corresponding to the permuted tokens. If provided, the unpermuted tokens will be merged with their respective probabilities. The shape is $[num\_tokens, topk]$, where *num_tokens* and *topk* are positive integers. Default: None .
- **padded_mode** (*bool*, *optional*) –If True, indicating the indices are padded to denote selected tokens per expert. Default: False .
- **restore_shape** (*Union[tuple[int], list[int]]*, *optional*) –The input shape before permutation, only used in padding mode. Default: None .

Returns

Tensor, with the same dtype as *permuted_tokens*. If *padded_mode* is `False`, the shape will be [*num_tokens*, *hidden_size*]. If *padded_mode* is `True`, the shape will be specified by *restore_shape*.

Raises

- **TypeError** –If *permuted_tokens* is not a Tensor.
- **ValueError** –Only supported when *padded_mode* is `False`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> permuted_token = Tensor([
...                          [1, 1, 1],
...                          [0, 0, 0],
...                          [0, 0, 0],
...                          [3, 3, 3],
...                          [2, 2, 2],
...                          [1, 1, 1],
...                          [2, 2, 2],
...                          [3, 3, 3]], dtype=mindspore.bfloat16)
>>> sorted_indices = Tensor([0, 6, 7, 5, 3, 1, 2, 4], dtype=mindspore.int32)
>>> out = ops.moe_token_unpermute(permuted_token, sorted_indices)
>>> out.shape
(8, 3)
```

mindspore.ops.incre_flash_attention

mindspore.ops.**incre_flash_attention**(*query*, *key*, *value*, *attn_mask=None*, *actual_seq_lengths=None*, *pse_shift=None*, *dequant_scale1=None*, *quant_scale1=None*, *dequant_scale2=None*, *quant_scale2=None*, *quant_offset2=None*, *antiquant_scale=None*, *antiquant_offset=None*, *block_table=None*, *num_heads=1*, *input_layout='BSH'*, *scale_value=1.0*, *num_key_value_heads=0*, *block_size=0*, *inner_precise=1*, *kv_padding_size=None*)

The interface for incremental inference.

- B: Batch size
- N: Num of attention heads
- kvN: Num of *key* / *value* heads
- S: Sequence length
- D: Head dim
- H: Hidden layer size
- kvH: Hidden size of *key* / *value*

where $H = N \times D$, $kvH = kvN \times D$.

Self attention constructs an attention model based on the relationship between input samples themselves. The principle is to assume that there is a length of the input sample sequence x of n, and each element of x is a d dimensional vector, which can be viewed as

a token embedding. This sequence can be transformed through 3 weight matrices to obtain 3 matrices with dimensions of $n \times d$. The self attention calculation formula is defined as:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d}})V$$

where the product of Q and K^T represents the attention of input x. To avoid the value becoming too large, it is usually scaled by dividing it by the square root of d and perform softmax normalization on each row, yields a matrix of $n \times d$ after multiplying V.

Note:

- If there is no input parameter and no default value, `None` needs to be passed.
- The shape of the tensor corresponding to the *key* and *value* parameters needs to be completely consistent.
- *N* of parameter *query* is equal with *num_heads*. *N* of parameter *key* and parameter *value* is equal with *num_key_value_heads*. *num_heads* is a multiple of *num_key_value_heads*.
- Quantization
  - When the data type of *query*, *key*, and *value* is float16 and the data type of output is int8, the input parameter *quant_scale2* is required and *quant_offset2* is optional.
  - When *antiquant_scale* exists, *key* and *value* need to be passed by int8. *antiquant_offset* is optional.
  - The data type of *antiquant_scale* and *antiquant_offset* should be consistenct with that of *query*.
- *pse_shift*
  - The *pse_shift* data type needs to be consistent with *query*, and only supports D-axis alignment, which means that the D-axis can be divided by 16.
- Page attention:
  - The necessary condition for enabling page attention is that the *block_table* exists, and the *key* and *value* are arranged in a contiguous memory according to the index in the *block_table*. The support dtype for *key* and *value* is float16/bfloat16/int8.
  - In the enabling scenario of page attention, 16 alignment is required when input types of *key* and *value* are float16/bfloat16, and 32 alignment is required when input dtype of *key* and *value* is int8. It is recommended to use 128.
  - The maximum max_block_num_per_seq currently supported by blocktable is 16k, and exceeding 16k will result in interception and error messages; If you encounter *S* being too large and causing max_block_num_per_seq to exceed 16k, you can increase the *block_size* to solve the problem.
  - The multiplication of all dimensions of the shape of the parameters *key* and *value* in the page attention scenario cannot exceed the representation range of int32.
  - When performing per-channel post quantization, page attention cannot be enabled simultaneously.
- *kv_padding_size*:
  - The calculation formula for the starting point of KV cache transfer is $S - kv\_padding\_size - actual\_seq\_lengths$. The calculation formula for the transfer endpoint of KV cache is $S - kv\_padding\_size$. When the starting or ending point of the KV cache transfer is less than 0, the returned data result is all 0.
  - When *kv_padding_size* is less than 0, it will be set to 0.
  - *kv_padding_size* needs to be enabled together with the *actual_seq_lengths* parameter, otherwise it is considered as the KV right padding scene.
  - It needs to be enabled together with the atten_mask parameter and ensure that the meaning of atten_mask is correct, that is, it can correctly hide invalid data. Otherwise, it will introduce accuracy issues.

– *kv_padding_size* does not support page attention scenarios.

Warning: Only support on Atlas A2 training series.

Parameters

- **query** (Tensor) –The query tensor with data type of float16 or bfloat16. The shape is $(B, 1, H)$ / $(B, N, 1, D)$.
- **key** (*Union[tuple, list]*) –The key tensor with data type of float16 or bfloat16 or int8. The shape is (B, S, kvH) / (B, kvN, S, D).
- **value** (*Union[tuple, list]*) –The value tensor with data type of float16 or bfloat16 or int8. The shape is (B, S, kvH) / (B, kvN, S, D).
- **attn_mask** (Tensor, *optional*) –The attention mask tensor with data type of bool or int8 or uint8. The shape is (B, S) / $(B, 1, S)$ / $(B, 1, 1, S)$. Default: None.
- **actual_seq_lengths** (*Union[Tensor, tuple[int], list[int]], optional*) –Describe actual sequence length of each input with data type of int64. The shape is $(B,)$. Default: None.
- **pse_shift** (Tensor, *optional*) –The position encoding tensor with data type of float16 or bfloat16. Input tensor of shape $(1, N, 1, S)$ / $(B, N, 1, S)$. Default: None.
- **dequant_scale1** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of uint64 or float32. It is disable now. Default: None.
- **quant_scale1** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of float32. It is disable now. Default: None.
- **dequant_scale2** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of uint64 or float32. It is disable now. Default: None.
- **quant_scale2** (Tensor, *optional*) –Post Quantitative parametor, the tensor with data type of float32. The shape is $(1,)$. Default: None.
- **quant_offset2** (Tensor, *optional*) –Post Quantitative parametor, the tensor with data type of float32. The shape is $(1,)$. Default: None.
- **antiquant_scale** (Tensor, *optional*) –Pseudo Quantitative parametor, the tensor with data type of float16 or bfloat16. The shape is $(2, kvN, 1, D)$ when input_layout is 'BNSD' or $(2, kvH)$ when input_layout is 'BSH'. Default: None.
- **antiquant_offset** (Tensor, *optional*) –Pseudo Quantitative parametor, the tensor with data type of float16 or bfloat16. The shape is $(2, kvN, 1, D)$ when input_layout is 'BNSD' or $(2, kvH)$ when input_layout is 'BSH'. Default: None.
- **block_table** (Tensor, *optional*) –The tensor with data type of int32. The shape is $(B, max\_block\_num\_per\_seq)$, where $max\_block\_num\_per\_seq = ceil(\frac{max(actual\_seq\_length)}{block\_size})$. Default: None.
- **num_heads** (*int, optional*) –The number of heads. Default: 1.
- **input_layout** (*str, optional*) –The data layout of the input qkv, support 'BSH' and 'BNSD'. Default 'BSH'.
- **scale_value** (*double, optional*) –The scale value indicating the scale coefficient, which is used as the scalar of Muls in the calculation. Default: 1.0.
- **num_key_value_heads** (*int, optional*) –Head numbers of *key*/*value* which are used in GQA algorithm. The value 0 indicates if the *key* and *value* have the same head nums, use numHeads. Default: 0.

- **block_size** (*int, optional*) –The maximum number of tokens stored in each block of KV in page attention. Default: `0`.
- **inner_precise** (*int, optional*) –An int number from {0, 1} indicates computing mode. `0` for high precision mode for float16 dtype. `1` for high performance mode. Default: `1`.
- **kv_padding_size** (`Tensor, optional`) –The tensor with data type of int64. The range of values is $0 \leq kv\_padding\_size \leq S - max(actual\_seq\_length)$. The shape is () or (1,). Default: `None`.

Returns

attention_out (Tensor), the shape is $(B, 1, H)$ / $(B, N, 1, D)$.

Raises

- **TypeError** –dtype of *query* is not float16 or bfloat16.
- **TypeError** –*key* and *value* don't have the same dtype.
- **TypeError** –dtype of *attn_mask* is not bool, int8 or uint8.
- **TypeError** –dtype of *pse_shift* is not bfloat16 or float16.
- **TypeError** –*scale_value* is not a double number.
- **TypeError** –*input_layout* is not a string.
- **TypeError** –*num_key_value_heads* or *num_heads* is not an int.
- **TypeError** –*inner_precise* is not an int.
- **TypeError** –*quant_scale1* is not Tensor of type float32.
- **TypeError** –*quant_scale2* is not Tensor of type float32.
- **TypeError** –*quant_offset2* is not Tensor of type float32.
- **ValueError** –size of *actual_seq_lengths* is not 1 or B.
- **ValueError** –*input_layout* is a string but of *(BSH)* or *(BNSD)*.
- **ValueError** –*num_heads* is not divisible by Q_H.
- **ValueError** –*num_heads* is not divisible by *num_key_value_heads*.
- **RuntimeError** –*num_heads* is not greater than 0.
- **RuntimeError** –*attn_mask* shape is not valid.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import ops
>>> from mindspore.common import Tensor
>>> from mindspore.common import dtype as mstype
>>> import numpy as np
>>> B, N, S, D, kvN = 1, 4, 10, 128, 1
>>> query = Tensor(np.random.randn(B, 1, N * D), mstype.float16)
>>> key = [Tensor(np.random.randn(B, S, kvN * D), mstype.float16)]
>>> value = [Tensor(np.random.randn(B, S, kvN * D), mstype.float16)]
>>> ifa_ms = ops.incre_flash_attention
```

(continues on next page)

(continued from previous page)

```
>>> attn_out = ifa_ms(query, key, value, num_heads=N, num_key_value_heads=kvN)
>>> attn_out
Tensor(shape=[1, 1, 512], dtype=Float16, value=
[[[ 1.6104e+00,  7.3438e-01,  1.0684e+00 ... -8.7891e-01,  1.7695e+00,  1.0264e+00]]])
```

mindspore.ops.prompt_flash_attention

mindspore.ops.**prompt_flash_attention**(*query*, *key*, *value*, *attn_mask=None*, *actual_seq_lengths=None*, *actual_seq_lengths_kv=None*, *pse_shift=None*, *deq_scale1=None*, *quant_scale1=None*, *deq_scale2=None*, *quant_scale2=None*, *quant_offset2=None*, *num_heads=1*, *scale_value=1.0*, *pre_tokens=2147483647*, *next_tokens=0*, *input_layout='BSH'*, *num_key_value_heads=0*, *sparse_mode=0*, *inner_precise=1*)

The interface for fully inference.

- B: Batch size
- N: Num of attention heads
- S: Sequence length
- D: Head dim
- H: Hidden layer size

Self attention constructs an attention model based on the relationship between input samples themselves. The principle is to assume that there is an input sample sequence x of length n, and each element of x is a d dimensional vector, which can be viewed as a token embedding. This sequence can be transformed through 3 weight matrices to obtain 3 matrices with dimensions of $n \times d$.

The self attention calculation formula is defined as:

$$Attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d}})V$$

where the product of Q and K^T represents the attention of input x. To avoid the value becoming too large, it is usually scaled by dividing it by the square root of d and perform softmax normalization on each row, yields a matrix of $n \times d$ after multiplying V.

Warning:

- Support dtype of float16 for *attn_mask* will be deprecated in the future.
- When *sparse_mode* is 2, 3 or 4, the shape of *attn_mask* must be $(2048, 2048)$ / $(B, 1, 2048, 2048)$ / $(1, 1, 2048, 2048)$.

Note:

- Maximum Support for each axis
  - Supports B-axis values less than or equal to 65536 (64k). When the input type includes int8 with D-axis not aligned to 32, or the input type is float16 or bfloat16 with D-axis not aligned to 16, the B-axis supports up to 128 only.
  - Supports N-axis values less than or equal to 256.
  - Supports S-axis values less than or equal to 20971520 (20M).
  - Supports D-axis values less than or equal to 512.
- Quantization

  - int8 Input, int8 Output: Parameters *deq_scale1*, *quant_scale1*, *deq_scale2*, and *quant_scale2* must all be provided. *quant_offset2* is optional (default is 0 if not provided).
  - int8 Input, float16 Output: Parameters *deq_scale1*, *quant_scale1*, and *deq_scale2* must all be provided. If *quant_offset2* or *quant_scale2* is provided (i.e., not null), it will result in an error.
  - float16 or bfloat16 Input, int8 Output: Parameter *quant_scale2* must be provided. *quant_offset2* is optional (default is 0 if not provided). If *deq_scale1*, *quant_scale1*, or *deq_scale2* is provided (i.e., not null), it will result in an error.
  - int8 Output:
    * *quant_scale2* and *quant_offset2* in per-channel format do not support scenarios with left padding, Ring Attention, or non-32-byte aligned D-axis.
    * In GE mode: *quant_scale2* and *quant_offset2* in per-tensor format do not support scenarios with non-32-byte aligned D-axis.
    * Does not support sparse as band and *pre_tokens*/*next_tokens* being negative.
  - *quant_scale2* and *quant_offset2* can be bfloat16 only when *query* is bfloat16.

- Other Usage Caveats:
  - *N* of parameter *query* must be equal to *num_heads*. *N* of parameter *key* and parameter *value* must be equal to *num_key_value_heads*.
  - *num_heads* must be divisible by *num_key_value_heads* and *num_heads* divided by *num_key_value_heads* can not be greater than 64.
  - When *query* dtype is bfloat16, D axis should align with 16.
  - Each element of *actual_seq_lengths* must not exceed q_S and element of *actual_seq_lengths_kv* must not exceed kv_S.

Warning: Only support on Atlas A2 training series.

Parameters

- **query** (Tensor) −The query tensor with data type of int8, float16 or bfloat16. The shape is (B, q_S, q_H) / *(B, q_N, q_S, q_D)*.
- **key** (Tensor) −The key tensor with the same dtype as *query*. The shape is (B, kv_S, kv_H) / *(B, kv_N, kv_S, kv_D)*.
- **value** (Tensor) −The value tensor with the same dtype as *query*. The shape is (B, kv_S, kv_H) / *(B, kv_N, kv_S, kv_D)*.
- **attn_mask** (Tensor, *optional*) −For each element, 0/False indicates retention and 1/True indicates discard. If *sparse_mode* is 0 or 1: the shape is (q_S, kv_S) / (B, q_S, kv_S) / $(1, q_S, kv_S)$ / $(B, 1, q_S, kv_S)$ / $(1, 1, q_S, kv_S)$. If *sparse_mode* is 2, 3 or 4, the shape is $(2048, 2048)$ / $(1, 2048, 2048)$ / $(1, 1, 2048, 2048)$. Default: None.
- **actual_seq_lengths** (*Union[Tensor, tuple[int], list[int]], optional*) −Describe actual sequence length of each batch of *query* with data type of int64. The shape is $(B,)$ and every element should be positive integer. Default: None.
- **actual_seq_lengths_kv** (*Union[Tensor, tuple[int], list[int]], optional*) −Describe actual sequence length of each batch of *key* or *value* with data type of int64. The shape is $(B,)$ and every element should be positive integer. Default: None.
- **pse_shift** (Tensor, *optional*) −The position encoding tensor with data type of float16 or bfloat16. Input tensor of shape (B, N, q_S, kv_S) / $(1, N, q_S, kv_S)$. Default: None.

  - q_S must be greater than or equal to the query's S length, and kv_S must be greater than or equal to the key's S length.'
  - If *pse_shift* has dtype float16, *query* should have dtype float16 or int8, in which case high precision mode is enabled automatically.
  - If *pse_shift* has dtype bfloat16, *query* should have dtype bfloat16.
- **deq_scale1** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of uint64 or float32. Input Tensor of shape $(1,)$. Default: None.
- **quant_scale1** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of float32. Input Tensor of shape $(1,)$. Default: None.
- **deq_scale2** (Tensor, *optional*) –Quantitative parametor, input Tensor of shape $(1,)$ and it has the same dtype as *deq_scale1*. Default: None.
- **quant_scale2** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of float32 or bfloat16. The suggested shape is $(1,)$ / $(1, 1, q_H)$ / $(q_H,)$ when output layout is BSH, $(1,)$ / $(1, q_N, 1, D)$ / $(q_N, D) when layout is BNSD. Default$:.
- **quant_offset2** (Tensor, *optional*) –Quantitative parametor, the tensor with data type of float32 or bfloat16. It has the same dtype and shape as *quant_scale2*. Default: None.
- **num_heads** (*int*, *optional*) –The number of heads. It is an integer in range [0, 256]. Default: 1.
- **scale_value** (*double*, *optional*) –The scale value indicating the scale coefficient, which is used as the scalar of Muls in the calculation. Default: 1.0.
- **pre_tokens** (*int*, *optional*) –For sparse cumputing, indicating the number of previous tokens Attention needs to associated with. Default: 2147483647.
- **next_tokens** (*int*, *optional*) –For sparse cumputing, indicating the number of next tokens Attention needs to associated with. Default: 0.
- **input_layout** (*str*, *optional*) –the data layout of the input qkv, support *(BSH)* and *(BNSD)*. Default BSH.
- **num_key_value_heads** (*int*, *optional*) –An int indicates head numbers of key/value which are used in GQA algorithm. The value 0 indicates if the key and value have the same head nums, use *num_heads*. It it is specified(not 0), it must be a factor of *num_heads* and it must be equal to kv_n. Default: 0.
- **sparse_mode** (*int*, *optional*) –An int specifies sparse mode, can be int from {0, 1, 2, 3, 4}. Default: 0.
  - sparseMode = 0: If *attn_mask* is a null pointer, *pre_tokens* and *next_tokens* inputs are ignored (internally set to INT_MAX).
  - sparseMode = 2, 3, 4: *attn_mask* shape must be (S, S) or $(1, S, S)$ or $(1, 1, S, S)$, with S fixed at 2048. User must ensure that *attn_mask* is lower triangular. If not provided or incorrect shape, it will result in an error.
  - sparseMode = 1, 2, 3: Ignores *pre_tokens*, *next_tokens* inputs and sets values according to specific rules.
  - sparseMode = 4: *pre_tokens* and *next_tokens* must be non-negative.
- **inner_precise** (*int*, *optional*) –An int number from {0, 1} indicates computing mode. 0 for high precision mode for float16 dtype. 1 for high performance mode. Default: 1.

Returns

attention_out (Tensor) - Output tensor, has the same shape as *query* of (B, q_S, q_H) / (B, q_N, q_S, q_D). Output dtype is determined by multiple factors, please refer to Note above for details.

Raises

- **TypeError** –Dtype of *query* is not int8, float16 or bfloat16.
- **TypeError** –*query*, *key* and *value* don't have the same dtype.

- **TypeError** – Dtype of *attn_mask* is not bool, int8 or uint8.
- **TypeError** – Dtype of *pse_shift* is not bfloat16 or float16.
- **TypeError** – *scale_value* is not a double number.
- **TypeError** – *input_layout* is not a string.
- **TypeError** – *num_key_value_heads* is not an int.
- **TypeError** – *sparse_mode* is not an int.
- **TypeError** – *sparse_inner_precisemode* is not an int.
- **TypeError** – *quant_scale1* is not Tensor of type float32.
- **TypeError** – *deq_scale1* is not Tensor of type uint64 or float32.
- **TypeError** – *quant_scale2* is not Tensor of type float32.
- **TypeError** – *deq_scale2* is not Tensor of type uint64 or float32.
- **TypeError** – *quant_offset2* is not Tensor of type float32.
- **ValueError** – *input_layout* is a string but of *(BSH)* or *(BNSD)*.
- **RuntimeError** – *num_heads* is not divisible by *num_key_value_heads*.
- **RuntimeError** – *num_heads* is not greater than 0.
- **RuntimeError** – *num_key_value_heads* is not greater than or equal to 0.
- **RuntimeError** – kv_n is not equal to *num_key_value_heads*.
- **RuntimeError** – *attn_mask* shape is not valid.
- **RuntimeError** – *sparse_mode* is specified but is not 0, 1, 2, 3 or 4.
- **RuntimeError** – *query* dtype is bfloat16 and D axis is not aligned with 16.
- **RuntimeError** – *input_layout* is BSH and kv_h is not divisible by *num_key_value_heads*.
- **RuntimeError** – D-axis of *query*, *key* and *value* is not the same.
- **RuntimeError** – In post quant per-channel scenario, D-axis is not 32 Byte aligned.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> B = 1
>>> N = 16
>>> S = 256
>>> D = 16
>>> query = Tensor(np.ones((B, N, S, D), dtype=np.float16))
>>> key = Tensor(np.ones((B, N, S, D), dtype=np.float16))
>>> value = Tensor(np.ones((B, N, S, D), dtype=np.float16))
>>> out = ops.prompt_flash_attention(query, key, value, None, None, None, None, None, None,␣
↪None, None,
...                                  None, N, input_layout='BNSD')
```

(continues on next page)

(continued from previous page)

```
>>> print(out.shape)
(1, 16, 256, 16)
```

mindspore.ops.flash_attention_score

mindspore.ops.**flash_attention_score**(*query*, *key*, *value*, *head_num*, *real_shift=None*, *drop_mask=None*, *padding_mask=None*, *attn_mask=None*, *prefix=None*, *actual_seq_qlen=None*, *actual_seq_kvlen=None*, *keep_prob=1.0*, *scalar_value=1.0*, *pre_tokens=2147483647*, *next_tokens=2147483647*, *inner_precise=0*, *input_layout='BSH'*, *sparse_mode=0*)

Implement self-attention calculations in training scenarios.

- B: Batch size. Value range 1 to 2k.
- S1: Sequence length of *query*. Value range 1 to 512k.
- S2: Sequence length of *key* and *value*. Value range 1 to 512k.
- N1: Num heads of *query*. Value range 1 to 256.
- N2: Num heads of *key* and *value*, and N2 must be a factor of N1.
- D: Head size. The value ranges is a multiple of 16, with the max value of 512.
- H1: Hidden size of *query*, which equals to N1 * D.
- H2: Hidden size of *key* and *value*, which equals to N2 * D.

The self attention calculation formula is defined as:

$$\text{attention\_out} = \text{Dropout}\left(\text{Softmax}\left(\text{Mask}(\text{scale} * \left(\text{query} * \text{key}^{\top}\right) + \text{pse}\right), \text{atten\_mask}), \text{keep\_prob}\right) * \text{value}$$

Warning:

- This is an experimental API that is subject to change or deletion.
- Only support on Atlas A2 training series.

Parameters

- **query** (Tensor) −The query tensor. Input tensor of shape $(B, S1, H1)$, $(B, N1, S1, D)$, $(S1, B, H1)$, $(B, S1, N1, D)$ or $(T1, N1, D)$. The supported dtype is float16 and bfloat16.
- **key** (Tensor) −The key tensor with the same dtype as *query*. Supported shape: $(B, S2, H2)$, $(B, N2, S2, D)$, $(S2, B, H2)$, $(B, S2, N2, D)$ or $(T2, N2, D)$.
- **value** (Tensor) −The value tensor with the same dtype and shape as *key*.
- **head_num** (*int*) −The head num of *query*, equal to N1.
- **real_shift** (Tensor, *optional*) −The position embedding code which is also known as pse, it has the same dtype as *query*. Default: None. If S is greater than 1024 and the mask of the lower triangle is used, only the inverse 1024 lines of the lower triangle is used for memory optimization. Input tensor of shape $(B, N1, S1, S2)$, $(1, N1, S1, S2)$, $(B, N1, 1024, S2)$, $(1, N1, 1024, S2)$.
  - ALiBi scenario: *real_shift* must meet the ALiBi rule, and sparse_mode is 2 or 3 for the lower triangle. In this scenario, *real_shift* is $(B, N1, 1024, S2)$, $(1, N1, 1024, S2)$.

  - Non-ALiBi scenario: *real_shift* is $(B, N1, S1, S2)$, $(1, N1, S1, S2)$.
  - input_layout is TND: shape should be $(B, N1, 1024, S2)$ and $(1, N1, 1024, S2)$.

- **drop_mask** (Tensor, *optional*) –The dropout mask tensor of uint8. Input tensor of shape $(B, N1, S1, S2//8) or None$. *S2* is a multiple of 8 when not None. Default: None.
- **padding_mask** (Tensor, *optional*) –Reserved parameter. Not implemented yet. Default: None.
- **attn_mask** (Tensor, *optional*) –The attention mask tensor of bool or uint8. For each element, 0/False indicates retention and 1/True indicates discard. Input tensor of shape $(B, N1, S1, S2)$, $(B, 1, S1, S2)$, $(S1, S2)$ or $(2048, 2048)$. Default: None.
  - In compression scenario, *sparse_mode* is 2, 3, or 4, *attn_mask* must be $(2048, 2048)$.
  - When *sparse_mode* is 5, *attn_mask* should be $(B, N1, S1, S2)$, $(B, 1, S1, S2)$.
  - When *sparse_mode* is 0 and 1, *attn_mask* should be $(B, N1, S1, S2)$, $(B, 1, S1, S2)$, $(S1, S2)$.
- **prefix** (*Union[Tensor, tuple[int], list[int]], optional*) –N value of each Batch in the prefix sparse calculation scenario. Input tensor of shape $(B,)$. B max value 32. Not none only when *sparse_mode* is 5. Default: None. If S1 > S2, N ranges from 0 to S2. If S1 <= S2, N ranges from S2 - S1 to S2.
- **actual_seq_qlen** (*Union[Tensor, tuple[int], list[int]], optional*) –Size of query corresponding to each batch, array with increasing values and the last value equal to T1. Default: None.
- **actual_seq_kvlen** (*Union[Tensor, tuple[int], list[int]], optional*) –Size of key and value corresponding to each batch, array with increasing values and the last value equal to T2. Default: None.
- **keep_prob** (*double, optional*) –The keep probability of dropout. Value range is (0.0, 1.0]. When *keep_prob* is 1.0, *drop_mask* should be None. Default: 1.0.
- **scalar_value** (*double, optional*) –The scale factor of score. Generally, the value is 1.0 / (D ** 0.5). Default: 1.0.
- **pre_tokens** (*int, optional*) –Parameter for sparse computation, represents how many tokens are counted forward. When *sparse_mode* is set to 1, 2, 3, or 5, this parameter does not take effect. Default: 2147483647.
- **next_tokens** (*int, optional*) –Parameter for sparse computation, represents how many tokens are counted backward. When *sparse_mode* is set to 1, 2, 3, or 5, this parameter does not take effect. Default: 2147483647. The value of *pre_tokens* corresponds to S1, and the value of *next_tokens* corresponds to S2. They define the valid area on the *attn_mask* matrix. It must ensure that the band is not empty. The following values are not allowed:
  - pre_tokens < 0 and next_tokens < 0.
  - (pre_tokens < 0 and next_tokens >= 0) and (next_tokens < abs(pre_tokens) or abs(pre_tokens) >= S2).
  - (pre_tokens >= 0 and next_tokens < 0) and (abs(next_tokens) > pre_tokens or abs(next_tokens) >= S1).
- **inner_precise** (*int, optional*) –The parameter is reserved and not implemented yet. Default:0.
- **input_layout** (*str, optional*) –Specifies the layout of input *query*, *key* and *value*. The value can be "BSH", "BNSD", "SBH", "BSND" or "TND". "TND" is an experimental format. Default: "BSH". When input_layout is "TND", the following restrictions must be met. Assume there are two lists that represent the length of the input sequence: list_seq_q and list_seq_k. Each value in the list indicates the length of the sequence in the batch. For example, list_seq_q = [4, 2, 6], list_seq_k = [10, 3, 9]. The element of list indicate S. T1 is sum(list_seq_q) = 12, T2 is sum(list_seq_k) = 22. max_seqlen_q = max(list_seq_q), max_seqlen_k = max(list_seq_k). qk_pointer = sum(list_seq_q * list_seq_k), which is the sum of the element multiplication.
  - The lengths of two lists must be the same, and size of list is batch. batch is less than or equal to 1024.
  - When *input_layout* is "TND", *actual_seq_qlen* and *actual_seq_kvlen* must be not none. Otherwise, they are none.
  - The *actual_seq_qlen* and *actual_seq_kvlen* are the cumulative sum of sequence of key/value, so they must be non-decreasing.

  - If *real_shift* is not none, list_seq_q and list_seq_k must be same. The maximum value of list_seq_q and list_seq_k is greater than 1024. *real_shift* should be $(B, N1, 1024, S2)$ and $(1, N1, 1024, S2)$, and S2 is equal to max_seqlen_k.
  - *attn_mask* must be a lower trianglar matrix, so *sparse_mode* should be 2 or 3. The shape of *attn_mask* should be $(2048, 2048)$.
  - The shape of *drop_mask* is $(qk\_pointer * N1//8,)$.
  - *prefix* is none.
  - *next_tokens* is 0, and *pre_tokens* is not less than max_seqlen_q.
  - When *sparse_mode* is 3, S1 of each batch should be less than or equal to S2.
  - 0 should not exist in list_seq_k.
- **sparse_mode** (*int, optional*) –Indicates sparse mode. Default: `0`.
  - 0: Indicates the defaultMask mode. If *attn_mask* is not passed, the mask operation is not performed, *next_tokens* and *pre_tokens* (internally assigned as INT_MAX) are ignored. If passed in, the full *attn_mask* matrix (S1 * S2) needs to be passed in, indicating that the part between *next_tokens* and *pre_tokens* needs to be calculated.
  - 1: Represents allMask, that is, passing in the complete *attn_mask* matrix.
  - 2: Representing the leftUpCausal mode corresponds to the lower triangle scenario divided by the left vertex, and the optimized *attn_mask* matrix (2048*2048) is required.
  - 3: Representing the rightDownCausal model corresponds to the lower triangle scene divided by the lower right vertex, and the optimized *attn_mask* matrix (2048*2048) is required.
  - 4: Represents the band scenario, that is, the part between counting *next_tokens* and *pre_tokens*, and the optimized *attn_mask* matrix (2048*2048) is required.
  - 5: Represents the prefix scenario, that is, on the basis of rightDownCasual, a matrix with length S1 and width N is added to the left side. The value of N is obtained by the new input prefix, and the N value of each Batch axis is different, not implemented yet.
  - 6: Represents the global scenario, not implemented yet.
  - 7: Represents the dilated scenario, not implemented yet.
  - 8: Represents the block_local scenario, not implemented yet.

Returns

attention_out (Tensor) - The output of attention, it has the same shape and dtype as *query*.

Raises

- **TypeError** –Dtype of *query* is not float16 or bfloat16.
- **TypeError** –*query*, *key* and *value* don't have the same dtype.
- **TypeError** –Dtype of *attn_mask* is not bool or uint8.
- **TypeError** –Dtype of *real_shift* has a different dtype as *query*.
- **TypeError** –*scalar_value* or *keep_prob* is not a double number.
- **TypeError** –*input_layout* is not a string.
- **TypeError** –*num_key_value_heads* is not an int.
- **TypeError** –*sparse_mode* is not an int.
- **TypeError** –*real_shift* is not Tensor type.
- **TypeError** –*drop_mask* is not Tensor type.

- **TypeError** – *padding_mask* is not Tensor type.
- **TypeError** – *attn_mask* is not Tensor type.
- **ValueError** – *input_layout* is a string but not valid.
- **RuntimeError** – *head_num* is not divisible by *N2*.
- **RuntimeError** – *head_num* is not greater than 0.
- **RuntimeError** – *attn_mask* shape is not valid.
- **RuntimeError** – The specified value of *sparse_mode* is invalid.
- **RuntimeError** – D-axis of *query*, *key* and *value* is not the same.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import mindspore.common.dtype as mstype
>>> import numpy as np
>>> from mindspore import ops, Tensor
>>> query = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> key = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> value = Tensor(np.ones([2, 4, 64]), dtype=mstype.float16)
>>> head_num = 4
>>> output = ops.flash_attention_score(query, key, value, head_num)
>>> print(output.shape)
(2, 4, 64)
```

mindspore.ops.rms_norm

mindspore.ops.**rms_norm**(*x*, *gamma*, *epsilon=1e-6*)

The RmsNorm(Root Mean Square Layer Normalization) operator is a normalization operation. Compared to LayerNorm, it retains scaling invariance and removes translation invariance. Its formula is:

$$y = \frac{x_i}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \varepsilon}}\gamma_i$$

Warning: This is an experimental API that is subject to change or deletion. This API is only supported in Atlas A2 training series for now.

Parameters

- **x** (Tensor) – Input data of RmsNorm. Support data type: float16, float32, bfloat16.
- **gamma** (Tensor) – Learnable parameter γ . Support data type: float16, float32, bfloat16.
- **epsilon** (*float, optional*) – A float number ranged in (0, 1] to prevent division by 0. Default value is *1e-6*.

Returns

- Tensor, denotes the normalized result, has the same type and shape as *x*.

- Tensor, with the float data type, denotes the reciprocal of the input standard deviation, used by gradient calculation.

Raises

- **TypeError** –If data type of *x* is not one of the following: float16, float32, bfloat16.
- **TypeError** –If data type of *gamma* is not one of the following: float16, float32, bfloat16.
- **TypeError** –If data type of *x* is not the same with the data type of *gamma*.
- **ValueError** –If *epsilon* is not a float between 0 and 1.
- **ValueError** –If the rank of *gamma* is lagger than the rank of *x*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1, 2, 3], [1, 2, 3]]), mindspore.float32)
>>> gamma = Tensor(np.ones([3]), mindspore.float32)
>>> y, rstd = ops.rms_norm(x, gamma)
>>> print(y)
[[0.46290997  0.92581993  1.3887299]
 [0.46290997  0.92581993  1.3887299]]
>>> print(rstd)
[[0.46290997]
 [0.46290997]]
```

mindspore.ops.unfold

mindspore.ops.**unfold**(*input*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Extracts sliding local blocks from a batched input tensor.

Consider a batched input tensor of shape $(N, C, *)$, where N is the batch dimension, C is the channel dimension, and $*$ represent arbitrary spatial dimensions. This operation flattens each sliding *Kernel_size*- sized block within the spatial dimensions of input x into a column (i.e., last dimension) of a 3-D output tensor of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where $C \times \prod(\text{kernel\_size})$ is the total number of values within each block (a block has $\prod(\text{kernel\_size})$ spatial locations, each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_{d} \left\lfloor \frac{\text{spatial\_size}[d] + 2 \times \text{pads}[d] - \text{dilations}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{strides}[d]} + 1 \right\rfloor,$$

where spatial_size is formed by the spatial dimensions of input x ($*$ above), and d is over all spatial dimensions.

Therefore, indexing *output* at the last dimension (column dimension) gives all values within a certain block.

The *dilation*, *padding* and *stride* arguments specify how the sliding blocks are retrieved.

Warning:

- The output is a 3-dimensional Tensor whose shape is $(N, C \times \prod(\text{kernel\_size}), L)$.
- This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –4-D Tensor, supported dtypes: float16, float32, float64, complex64 and complex128.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) –The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width.
- **dilation** (*Union[int, tuple[int], list[int]], optional*) –The dilation of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: 1 .
- **padding** (*Union[int, tuple[int], list[int]], optional*) –The pad of the window, that must be a tuple/list of one or two *int* for height and width. Default: 0 .
  - If one int, pad_height = pad_width.
  - If two int, pad_height = padding[0], pad_width = padding[1].
- **stride** (*Union[int, tuple[int], list[int]], optional*) –The stride of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: 1 .

Returns

A Tensor, with same type as *input* . And its shape is as described above.

Raises

- **TypeError** –If any data type of *kernel_size*, *stride*, *dilation*, *padding* is not int, tuple or list.
- **ValueError** –If *kernel_size*, *dilation*, *stride* value is not greater than zero or elements number more than *2*.
- **ValueError** –If *padding* value is less than zero.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.rand(4, 4, 32, 32), mindspore.float64)
>>> output = ops.unfold(x, kernel_size=3, dilation=1, stride=1)
>>> print(output.shape)
(4, 36, 900)
```

2.3.2 Loss Functions

API Name	Description	Supported Platforms
mindspore.ops.binary_cross_entropy	Computes the binary cross entropy(Measure the difference information between two probability distributions) between predictive value *logits* and target value *labels*.	Ascend GPU CPU
mindspore.ops.binary_cross_entropy_with_logits	Adds sigmoid activation function to input *input* as logits, and uses the given logits to compute binary cross entropy between the *input* and the *target*.	Ascend GPU CPU

continues on next page

Table 12 – continued from previous page

mindspore.ops.cosine_embedding_loss	CosineEmbeddingLoss creates a criterion to measure the similarity between two tensors using cosine distance.	Ascend GPU CPU
mindspore.ops.cross_entropy	The cross entropy loss between input and target.	Ascend GPU CPU
mindspore.ops.ctc_loss	Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.	Ascend GPU CPU
mindspore.ops.gaussian_nll_loss	Gaussian negative log likelihood loss.	Ascend GPU CPU
mindspore.ops.hinge_embedding_loss	Measures Hinge Embedding Loss given an input Tensor *intputs* and a labels Tensor *targets* (containing 1 or -1).	Ascend GPU CPU
mindspore.ops.huber_loss	Calculates the error between the predicted value and the target value, which has the best of both the loss of mindspore.ops.l1_loss() and the loss of mindspore.ops.mse_loss().	Ascend GPU CPU
mindspore.ops.kl_div	Computes the Kullback-Leibler divergence between the logits and the labels.	Ascend GPU CPU
mindspore.ops.l1_loss	Calculate the mean absolute error between the *input* value and the *target* value.	Ascend GPU CPU
mindspore.ops.margin_ranking_loss	MarginRankingLoss creates a criterion that measures the loss.	Ascend GPU CPU
mindspore.ops.mse_loss	Calculates the mean squared error between the predicted value and the label value.	Ascend GPU CPU
mindspore.ops.multi_margin_loss	Hinge loss for optimizing a multi-class classification.	Ascend GPU CPU
mindspore.ops.multilabel_margin_loss	Hinge loss for optimizing a multi-label classification.	Ascend GPU
mindspore.ops.multilabel_soft_margin_loss	Calculates the MultiLabelSoftMarginLoss.	Ascend GPU CPU
mindspore.ops.nll_loss	Gets the negative log likelihood loss between inputs and target.	Ascend GPU CPU
mindspore.ops.smooth_l1_loss	Computes smooth L1 loss, a robust L1 loss.	Ascend GPU CPU
mindspore.ops.soft_margin_loss	Calculate the soft margin loss of input and target.	Ascend GPU
mindspore.ops.triplet_margin_loss	TripletMarginLoss operation.	GPU

mindspore.ops.binary_cross_entropy

mindspore.ops.**binary_cross_entropy**(*logits*, *labels*, *weight=None*, *reduction='mean'*)

Computes the binary cross entropy(Measure the difference information between two probability distributions) between predictive value *logits* and target value *labels*.

Set *logits* as x, *labels* as y, output as $\ell(x, y)$, the weight of nth batch of binary cross entropy is w_n. Let,

$$L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

In which, L indicates the loss of all *batch_size*, l indicates the loss of one *batch_size*, and n indicates one *batch_size* in the $1 - N$ range. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Warning:

- The value of *logits* must range from *0* to *1*.

Parameters

- **logits** (Tensor) –The predictive value whose data type must be float16 or float32.
- **labels** (Tensor) –The target value which has the same shape and data type as *logits*. And the data type is float16 or float32.
- **weight** (Tensor, *optional*) –A rescaling weight applied to the loss of each batch element. Its shape must be able to broadcast to that of *logits* and *labels*. And it must have the same shape and data type as *logits*. Default: None . If set to None , the loss function will not consider any sample weights, and each sample will be treated as having equal importance when calculating the loss.
- **reduction** (*str*, *optional*) –Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the weighted mean of elements in the output.
  - 'sum': the output elements will be summed.

Returns

Tensor or Scalar. Returns Tensor that has the same dtype and shape as *logits* if *reduction* is 'none'. Otherwise, returns a scalar Tensor.

Raises

- **TypeError** –If *logits*, *labels* or *weight* is not a Tensor.
- **TypeError** –If dtype of *logits*, *labels* or *weight* (if given) is neither float16 nor float32.
- **ValueError** –If *reduction* is not one of 'none', 'mean' or 'sum'.
- **ValueError** –If shape of *labels* is not the same as *logits* or *weight* (if given).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([0.2, 0.7, 0.1]), mindspore.float32)
>>> labels = Tensor(np.array([0., 1., 0.]), mindspore.float32)
>>> weight = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = ops.binary_cross_entropy(logits, labels, weight)
>>> print(output)
0.38240486
```

mindspore.ops.binary_cross_entropy_with_logits

mindspore.ops.**binary_cross_entropy_with_logits**(*input*, *target*, *weight=None*, *pos_weight=None*, *reduction='mean'*)

Adds sigmoid activation function to input *input* as logits, and uses the given logits to compute binary cross entropy between the *input* and the *target*.

Sets input *input* as X, input *target* as Y, input *weight* as W, output as L. Then,

$$
\begin{aligned}
p_{ij} &= sigmoid(X_{ij}) = \frac{1}{1+e^{-X_{ij}}} \\
L_{ij} &= -[Y_{ij}log(p_{ij}) + (1 - Y_{ij})log(1 - p_{ij})]
\end{aligned}
$$

i indicates the i^{th} sample, j indicates the category. Then,

$$
\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}
$$

ℓ indicates the method of calculating the loss. There are three methods: the first method is to provide the loss value directly, the second method is to calculate the average value of all losses, and the third method is to calculate the sum of all losses.

This operator will multiply the output by the corresponding weight. The tensor $weight$ assigns different weights to each piece of data in the batch, and the tensor pos_weight adds corresponding weights to the positive examples of each category.

In addition, it can trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$
\begin{aligned}
p_{ij,c} &= sigmoid(X_{ij,c}) = \frac{1}{1+e^{-X_{ij,c}}} \\
L_{ij,c} &= -[P_c Y_{ij,c} * log(p_{ij,c}) + (1 - Y_{ij,c})log(1 - p_{ij,c})]
\end{aligned}
$$

where c is the class number (c>1 for multi-label binary classification, c=1 for single-label binary classification), n is the number of the sample in the batch and P_c is the weight of the positive answer for the class c. $P_c > 1$ increases the recall, $P_c < 1$ increases the precision.

Parameters

- **input** (Tensor) −Input *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **target** (Tensor) −Ground truth label, has the same shape as *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **weight** (Tensor, *optional*) −A rescaling weight applied to the loss of each batch element. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: None, it equals to *weight* is a Tensor whose value is 1.
- **pos_weight** (Tensor, *optional*) −A weight of positive examples. Must be a vector with length equal to the number of classes. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: None, it equals to *pos_weight* is a Tensor whose value is 1.
- **reduction** (*str, optional*) −Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the weighted mean of elements in the output.
  - 'sum': the output elements will be summed.

Returns

Tensor or Scalar, if *reduction* is `'none'`, it's a tensor with the same shape and type as input *input*. Otherwise, the output is a scalar.

Raises

- **TypeError** –If input *input*, *target*, *weight*, *pos_weight* is not Tensor.
- **TypeError** –If data type of input *input*, *target*, *weight*, *pos_weight* is not float16, float32 or bfloat16.
- **TypeError** –If data type of input *reduction* is not string.
- **ValueError** –If *weight* or *pos_weight* can not be broadcast to a tensor with shape of *input*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]), mindspore.float32)
>>> target = Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]), mindspore.float32)
>>> weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> pos_weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> output = ops.binary_cross_entropy_with_logits(input, target, weight, pos_weight)
>>> print(output)
0.3463612
```

mindspore.ops.cosine_embedding_loss

`mindspore.ops.`**`cosine_embedding_loss`**(*input1*, *input2*, *target*, *margin=0.0*, *reduction='mean'*)

CosineEmbeddingLoss creates a criterion to measure the similarity between two tensors using cosine distance.

Given two tensors $input1$, $input2$, and a Tensor label $target$ with values 1 or -1:

$$loss(input1, input2, target) = \begin{cases} 1 - cos(input1, input2), & \text{if } target = 1 \\ max(0, cos(input1, input2) - margin), & \text{if } target = -1 \end{cases}$$

Parameters

- **input1** (Tensor) –Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **input2** (Tensor) –Tensor of shape $(N, *)$, same shape and dtype as *input1*.
- **target** (Tensor) –Contains value 1 or -1. Suppose the shape of *input1* is $(x_1, x_2, x_3, ..., x_R)$, then the shape of *target* must be $(x_1, x_3, x_4, ..., x_R)$.
- **margin** (*float, optional*) –Should be in [-1.0, 1.0]. Default: `0.0`.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.

- `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar, if *reduction* is `"none"`, its shape is the same as *target*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If *margin* is not a float.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** – If *margin* is not in range [-1.0, 1.0].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> intput1 = Tensor(np.array([[0.3, 0.8], [0.4, 0.3]]), mindspore.float32)
>>> intput2 = Tensor(np.array([[0.4, 1.2], [-0.4, -0.9]]), mindspore.float32)
>>> target = Tensor(np.array([1, -1]), mindspore.int32)
>>> output = ops.cosine_embedding_loss(intput1, intput2, target)
>>> print(output)
0.0003425479
```

mindspore.ops.cross_entropy

`mindspore.ops.`**`cross_entropy`**(*input*, *target*, *weight=None*, *ignore_index=- 100*, *reduction='mean'*, *label_smoothing=0.0*)

The cross entropy loss between input and target.

The cross entropy support two kind of targets:

- Class indices (int) in the range $[0, C)$ where C is the number of classes, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}$$

  where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

  If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

- Probabilities (float) for each class, useful when labels beyond a single class per minibatch item are required, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

  where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **input** (Tensor) −(N) or (N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$. *input* is expected to be log-probabilities, data type must be float16 or float32.
- **target** (Tensor) −For class indices, tensor of shape (), (N) or $(N, d_1, d_2, ..., d_K)$, data type must be int32. For probabilities, tensor of shape $(C,)$, (N, C) or $(N, C, d_1, d_2, ..., d_K)$, data type must be float16 or float32 or float64.
- **weight** (Tensor) −A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$, data type must be float16 or float32. Default: `None` .
- **ignore_index** (*int*) −Specifies a target value that is ignored and does not contribute to the input gradient. Default: `-100` .
- **reduction** (*str, optional*) −Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **label_smoothing** (*float*) −Label smoothing values, a regularization tool used to prevent the model from overfitting when calculating Loss. The value range is [0.0, 1.0]. Default value: `0.0` .

Returns

Tensor, the computed loss value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> # Case 1: Indices labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.array([1, 0, 4]), ms.int32)
>>> output = ms.ops.cross_entropy(inputs, target)
>>> # Case 2: Probability labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> output = ms.ops.cross_entropy(inputs, target)
```

mindspore.ops.ctc_loss

mindspore.ops.**ctc_loss**(*log_probs*, *targets*, *input_lengths*, *target_lengths*, *blank=0*, *reduction='mean'*, *zero_infinity=False*)

Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.

CTC is a loss function in sequence labeling problems, which is mainly used to deal with the alignment of input and output labels in sequence labeling problems. While traditional sequence labeling algorithms require the input and output symbols to be perfectly aligned at each moment, CTC expands the label collection and adds empty elements. After labeling the sequence using the extended label set, all the prediction sequences that can be converted into real sequences by the mapping function are correct prediction results, that is, the predicted sequence can be obtained without data alignment processing. Its objective function is to maximize the sum of probabilities of all correct prediction sequences.

The CTC algorithm is proposed in Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks.

Parameters

- **log_probs** (Tensor) – A tensor of shape (T, N, C), where T is input length, N is batch size and C is number of classes (including blank).
- **targets** (Tensor) – Target sequences. A tensor of shape (N, S), where S is max target length.
- **input_lengths** (*Union(tuple, Tensor)*) – Lengths of the input. A tuple or Tensor of shape (N).
- **target_lengths** (*Union(tuple, Tensor)*) – Lengths of the target. A tuple or Tensor of shape (N).
- **blank** (*int, optional*) – The blank label. Default: `0` .
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **zero_infinity** (*bool, optional*) – Whether to set infinite loss and correlation gradient to 0. Default: `False` .

Returns

neg_log_likelihood (Tensor), A loss value with shape (N) , which is differentiable with respect to each input node.

log_alpha (Tensor), The probability of possible trace of input to target with shape $(N, T, 2 * S + 1)$.

Raises

- **TypeError** – If *zero_infinity* is not a bool, *reduction* is not string.
- **TypeError** – If the dtype of *log_probs* is not float or double.
- **TypeError** – If the dtype of *targets*, *input_lengths* or *target_lengths* is not int32 or int64.
- **ValueError** – If the rank of *log_probs* is not 3.
- **ValueError** – If the rank of *targets* is not 2.
- **ValueError** – If the shape of *input_lengths* does not match N. N is batch size of *log_probs* .
- **ValueError** – If the shape of *target_lengths* does not match N. N is batch size of *log_probs* .
- **ValueError** – If the value of *blank* is not in range [0, num_labels|C). C is number of classes of *log_probs* .
- **RuntimeError** – If any value of *input_lengths* is larger than T. T is the length of *log_probs*.

- **RuntimeError** –If any target_lengths[i] is not in range [0, input_length[i]].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> log_probs = Tensor(np.array([[[0.3, 0.6, 0.6]],
...                              [[0.9, 0.4, 0.2]]]).astype(np.float32))
>>> targets = Tensor(np.array([[0, 1]]), mstype.int32)
>>> input_lengths = Tensor(np.array([2]), mstype.int32)
>>> target_lengths = Tensor(np.array([1]), mstype.int32)
>>> loss, log_alpha = ops.ctc_loss(log_probs, targets, input_lengths,
...                                target_lengths, 0, 'mean', True)
>>> print(loss)
-2.2986124
>>> print(log_alpha)
[[[0.3       0.3             -inf      -inf      -inf]
  [1.2       1.8931472 1.2             -inf      -inf]]]
```

mindspore.ops.gaussian_nll_loss

mindspore.ops.**gaussian_nll_loss**(*x*, *target*, *var*, *full=False*, *eps=1e-6*, *reduction='mean'*)

Gaussian negative log likelihood loss.

The target values are considered to be samples from a Gaussian distribution, where the expectation and variance are predicted by a neural network. For *labels* modeled on a Gaussian distribution, *logits* to record expectations, and the variance *var* (elements are all positive), the calculated loss is:

$$\text{loss} = \frac{1}{2}\left(\log\left(\max\left(\text{var},\ \text{eps}\right)\right) + \frac{\left(\text{x} - \text{target}\right)^2}{\max\left(\text{var},\ \text{eps}\right)}\right) + \text{const.}$$

where eps is used for stability of log. When $full = True$, a constant will be added to the loss. If the shape of var and $logits$ are not the same (due to a homoscedastic assumption), their shapes must allow correct broadcasting.

Parameters

- **x** (Tensor) –Tensor of shape $(N, *)$ or $(*)$ where $*$ means any number of additional dimensions.
- **target** (Tensor) –Tensor of shape $(N, *)$ or $(*)$, same shape as the x, or same shape as the x but with one dimension equal to 1 (to allow broadcasting).
- **var** (Tensor) –Tensor of shape $(N, *)$ or $(*)$, same shape as x, or same shape as the x but with one dimension equal to 1, or same shape as the x but with one fewer dimension (to allow for broadcasting).
- **full** (*bool, optional*) –Include the constant term in the loss calculation. When $full = True$, the constant term will be $const = 0.5 * log(2\pi)$. Default: `False`.
- **eps** (*float, optional*) –Used to improve the stability of log function must be greater than 0. Default: `1e-6` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.

- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Returns

Tensor or Tensor scalar, the computed loss depending on $reduction$.

Raises

- **TypeError** – If *x*, *target* or *var* is not a Tensor.
- **TypeError** – If *full* is not a bool.
- **TypeError** – If *eps* is not a float.
- **ValueError** – If *eps* is not a float within (0, inf).
- **ValueError** – If *reduction* is not one of `"none"` , `"mean"` , `"sum"` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> import mindspore.common.dtype as mstype
>>> arr1 = np.arange(8).reshape((4, 2))
>>> arr2 = np.array([2, 3, 1, 4, 6, 4, 4, 9]).reshape((4, 2))
>>> x = Tensor(arr1, mstype.float32)
>>> var = Tensor(np.ones((4, 1)), mstype.float32)
>>> target = Tensor(arr2, mstype.float32)
>>> output = ops.gaussian_nll_loss(x, target, var)
>>> print(output)
1.4374993
```

Reference:

Nix, D. A. and Weigend, A. S., "Estimating the mean and variance of the target probability distribution", Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94), Orlando, FL, USA, 1994, pp. 55-60 vol.1, doi: 10.1109/ICNN.1994.374138.

mindspore.ops.hinge_embedding_loss

`mindspore.ops.`**`hinge_embedding_loss`**(*inputs*, *targets*, *margin=1.0*, *reduction='mean'*)

Measures Hinge Embedding Loss given an input Tensor *intputs* and a labels Tensor *targets* (containing 1 or -1).

The loss function for n-th sample in the mini-batch is

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

where $L = \{l_1, \ldots, l_N\}^\top$.

Parameters

- **inputs** (Tensor) –Predicted values, represented as *x* in the formula.
- **targets** (Tensor) –Label values, represented as *y* in the formula. Has the same shape as *inputs*, contains -1 or 1.
- **margin** (*float, int*) –Threshold defined by Hinge Embedding Loss *margin*. Represented as Δ in the formula. Default: `1.0`.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Tensor scalar, the computed loss depending on *reduction*.

Raises

- **TypeError** –If *inputs* is not a Tensor.
- **TypeError** –If *targets* is not a Tensor.
- **TypeError** –If *margin* is not a float or int.
- **ValueError** –If *targets* does not have the same shape as *inputs* or they could not broadcast to each other.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> import mindspore.common.dtype as mstype
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> arr1 = np.array([0.9, -1.2, 2, 0.8, 3.9, 2, 1, 0, -1]).reshape((3, 3))
>>> arr2 = np.array([1, 1, -1, 1, -1, 1, -1, 1, 1]).reshape((3, 3))
>>> logits = Tensor(arr1, mstype.float32)
>>> labels = Tensor(arr2, mstype.float32)
>>> loss = ops.hinge_embedding_loss(logits, labels, margin=1.0, reduction='mean')
>>> print(loss)
0.16666666
```

mindspore.ops.huber_loss

mindspore.ops.**huber_loss**(*input*, *target*, *reduction='mean'*, *delta=1.0*)

Calculates the error between the predicted value and the target value, which has the best of both the loss of *mindspore.ops.l1_loss()* and the loss of *mindspore.ops.mse_loss()*.

Assuming that the x and y are 1-D Tensor, length N, the *reduction* parameter is set to `'none'` then calculate the loss of x and y without dimensionality reduction. The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top$$

with

$$l_n = \begin{cases} 0.5 * (x_n - y_n)^2, & \text{if } |x_n - y_n| < delta; \\ delta * (|x_n - y_n| - 0.5 * delta), & \text{otherwise.} \end{cases}$$

where N is the batch size.

If *reduction* is "mean" or "sum", then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = "mean";} \\ \text{sum}(L), & \text{if reduction = "sum".} \end{cases}$$

Parameters

- **input** (Tensor) –Predicted value, Tensor of any dimension.
- **target** (Tensor) –Target value, has same dtype and shape as the *input* in common cases. However, when the shape of *target* is different from the shape of *input*, and they should be broadcasted to each other.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **delta** (*Union[int, float]*) –The threshold to change between two type of loss. The value must be greater than zero. Default: `1.0`.

Returns

Tensor or Scalar, if *reduction* is `'none'`, return a Tensor with same shape and dtype as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *input* or *target* is not a Tensor.
- **TypeError** –If dtype of *delta* is neither float nor int.
- **ValueError** –If *delta* is less than or equal to 0.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** –If *input* and *target* have different shapes and cannot be broadcasted to each other.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([1, 2, 10, 2], mindspore.float32)
>>> target = Tensor([1, 5, 1, 20], mindspore.float32)
>>> output = ops.huber_loss(x, target, reduction="mean", delta=2)
>>> print(output)
13.5
```

mindspore.ops.kl_div

mindspore.ops.**kl_div**(*logits*, *labels*, *reduction='mean'*)

Computes the Kullback-Leibler divergence between the logits and the labels.

For input tensors x and $target$ with the same shape, the updating formulas of KLDivLoss algorithm are as follows,

$$L(x, target) = target \cdot (\log target - x)$$

Then,

$$\ell(x, target) = \begin{cases} L(x, target), & \text{if reduction = 'none';} \\ \text{mean}(L(x, target)), & \text{if reduction = 'mean';} \\ \text{sum}(L(x, target))/x.\text{shape}[0], & \text{if reduction = 'batchmean';} \\ \text{sum}(L(x, target)), & \text{if reduction = 'sum'.} \end{cases}$$

where x represents *logits*. $target$ represents *labels*. $\ell(x, target)$ represents *output*.

Note:

- Currently it does not support float64 input on *Ascend*.
- The output aligns with the mathematical definition of Kullback-Leibler divergence only when *reduction* is set to `'batchmean'`.

Parameters

- **logits** (Tensor) −The input Tensor. The data type must be float16, float32 or float64.
- **labels** (Tensor) −The label Tensor which has the same shape and data type as *logits*.
- **reduction** (*str*) −Specifies the reduction to be applied to the output. Its value must be one of `'none'`, `'mean'`, `'batchmean'` or `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
  - `'batchmean'`: the summed output elements divided by batch size.

Returns

Tensor or Scalar, if *reduction* is `'none'`, then output is a tensor and has the same shape as *logits*. Otherwise, it is a scalar.

Raises

- **TypeError** –If *reduction* is not a str.
- **TypeError** –If neither *logits* nor *labels* is a Tensor.
- **TypeError** –If dtype of *logits* or *labels* is not the supported type.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([0.2, 0.7, 0.1]), mindspore.float32)
>>> labels = Tensor(np.array([0., 1., 0.]), mindspore.float32)
>>> output = mindspore.ops.kl_div(logits, labels, 'mean')
>>> print(output)
-0.23333333
```

mindspore.ops.l1_loss

`mindspore.ops.`**`l1_loss`**(*input*, *target*, *reduction='mean'*)

Calculate the mean absolute error between the *input* value and the *target* value.

Assuming that the x and y (predicted and target value) are 1-D Tensor, length N, *reduction* is set to `'none'`, then calculate the loss of x and y without dimensionality reduction.

The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with } l_n = |x_n - y_n|,$$

where N is the batch size.

If *reduction* is set to `'mean'` or `'sum'` , then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **input** (Tensor) –Predicted value, Tensor of any dimension.
- **target** (Tensor) –Target value, usually has the same shape as the *input*. If *input* and *target* have different shape, make sure they can broadcast to each other.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar, if *reduction* is `'none'`, return a Tensor with same shape and dtype as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *target* is not a Tensor.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor([[1, 2, 3], [4, 5, 6]], mstype.float32)
>>> target = Tensor([[6, 5, 4], [3, 2, 1]], mstype.float32)
>>> output = ops.l1_loss(x, target, reduction="mean")
>>> print(output)
3.0
```

mindspore.ops.margin_ranking_loss

`mindspore.ops.`**`margin_ranking_loss`**(*input1*, *input2*, *target*, *margin=0.0*, *reduction='mean'*)

MarginRankingLoss creates a criterion that measures the loss.

Given two tensors $input1$, $input2$ and a Tensor label $target$ with values 1 or -1, the operation is as follows:

$$\text{loss}(input1, input2, target) = \max(0, -target * (input1 - input2) + \text{margin})$$

Parameters

- **input1** (`Tensor`) – Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **input2** (`Tensor`) – Tensor of shape $(N, *)$, same shape and dtype as *input1*.
- **target** (`Tensor`) – Contains value 1 or -1. Suppose the shape of *input1* is $(x_1, x_2, x_3, ..., x_R)$, then the shape of *target* must be $(x_1, x_2, x_3, ..., x_R)$.
- **margin** (*`float, optional`*) – Specify the adjustment factor of the operation. Default: `0.0` .
- **reduction** (*`str, optional`*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar. if *reduction* is `'none'`, its shape is the same as *input1*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If *margin* is not a float.
- **TypeError** – If *input1*, *input2* or *target* is not a Tensor.
- **TypeError** – If the types of *input1* and *input2* are inconsistent.

- **TypeError** –If the types of *input1* and *target* are inconsistent.
- **ValueError** –If the shape of *input1* and *input2* are inconsistent.
- **ValueError** –If the shape of *input1* and *target* are inconsistent.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> input1 = Tensor(np.array([0.3864, -2.4093, -1.4076]), ms.float32)
>>> input2 = Tensor(np.array([-0.6012, -1.6681, 1.2928]), ms.float32)
>>> target = ops.Sign()(Tensor(np.array([-2, -2, 3]), ms.float32))
>>> output = ops.margin_ranking_loss(input1, input2, target)
>>> print(output)
1.2293333
```

mindspore.ops.mse_loss

`mindspore.ops.`**`mse_loss`**(*input*, *target*, *reduction='mean'*)

Calculates the mean squared error between the predicted value and the label value.

For detailed information, please refer to *mindspore.nn.MSELoss*.

Parameters

- **input** (Tensor) –Tensor of any dimension.
- **target** (Tensor) –The input label. Tensor of any dimension, same shape as the *input* in common cases. However, it supports that the shape of *input* is different from the shape of *target* and they should be broadcasted to each other.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor, loss of type float, the shape is zero if *reduction* is `'mean'` or `'sum'`, while the shape of output is the broadcasted shape if *reduction* is `'none'`.

Raises

- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.
- **ValueError** –If *input* and *target* have different shapes and cannot be broadcasted.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = ops.mse_loss(logits, labels, reduction='none')
>>> print(output)
[[0. 1. 4.]
 [0. 0. 1.]]
```

mindspore.ops.multi_margin_loss

mindspore.ops.**multi_margin_loss**(*input*, *target*, *p=1*, *margin=1*, *weight=None*, *reduction='mean'*)

Hinge loss for optimizing a multi-class classification.

Optimizes a multi-class classification hinge loss (margin-based loss) between input and output.

For each mini-batch sample, the loss in terms of the 1D input x and scalar output y is:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, \text{margin} - x[y] + x[i])^p}{\text{x.size}(0)}$$

where $i \in \{0, , x.size(0) - 1\}$ and $i \neq y$.

Parameters

- **input** (Tensor) –Input , with shape (N, C). Data type only support float32, float16 or float64. It is x in the above formula.
- **target** (Tensor) –Ground truth labels, with shape $(N,)$. Data type only support int64. The value of target should be non-negative, less than C. It is y in the above formula.
- **p** (*int, optional*) –The norm degree for pairwise distance. Should be 1 or 2. Default: `1` .
- **margin** (*int, optional*) –A parameter to change pairwise distance. Default: `1` .
- **weight** (Tensor, *optional*) –The rescaling weight to each class with shape $(C,)$. Data type only support float16, float32 or float64. Default: `None` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

- **outputs** - Tensor. If *reduction* is `'none'`, returns a Tensor with the same shape as *target*. Otherwise, it is a scalar.

Raises

- **TypeError** –If dtype of *p* or *target* is not int.
- **TypeError** –If dtype of *margin* is not int.
- **TypeError** –If dtype of *reduction* is not str.
- **TypeError** –If dtype of *input* is not float16, float or float64.

- **TypeError** –If dtype of *weight* and *input* is not the same.
- **ValueError** –If *p* is not 1 or 2.
- **ValueError** –If *reduction* is not one of {'none','sum','mean'}.
- **ValueError** –If shape[0] of *input* is not equal to shape[0] of *target*.
- **ValueError** –If shape[1] of *input* is not equal to shape[0] of *weight*.
- **ValueError** –If rank of *weight* is not 1 or rank of *target* is not 1 or *input* is not 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inputs = Tensor(np.ones(shape=[3, 3]), mindspore.float32)
>>> target = Tensor(np.array([1, 2, 1]), mindspore.int64)
>>> weight = Tensor(np.array([1, 1, 1]), mindspore.float32)
>>> output = ops.multi_margin_loss(inputs, target, weight=weight)
>>> print(output)
0.6666667
```

mindspore.ops.multilabel_margin_loss

`mindspore.ops.`**`multilabel_margin_loss`**(*input*, *target*, *reduction='mean'*)

Hinge loss for optimizing a multi-label classification.

Creates a criterion that optimizes a multi-label multi-classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 2D *Tensor* of target class indices). For each sample in the mini-batch:

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$, $y \in \{0, \cdots, \text{y.size}(0) - 1\}$, $0 \leq y[j] \leq \text{x.size}(0) - 1$, and $i \neq y[j]$ for all i and j. y and x must have the same size. The criterion only considers a contiguous block of non-negative targets that starts at the front. This allows for different samples to have variable amounts of target classes.

Parameters

- **input** (`Tensor`) –Predict data, x in the formula above. Tensor of shape (C) or (N, C), where N is the batch size and C is the number of classes. Data type must be float16 or float32.
- **target** (`Tensor`) –Ground truth data, y in the formula above, with the same shape as *input*, data type must be int32 and label targets padded by -1.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

- **outputs** (Union[Tensor, Scalar]) - The loss of MultilabelMarginLoss. If *reduction* is `"none"`, its shape is (N). Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *input* or *target* is not a Tensor.
- **TypeError** –If dtype of *input* is neither float16 nor float32.
- **TypeError** –If dtype of *target* is not int32.
- **ValueError** –If length of shape of *input* is neither 1 nor 2.
- **ValueError** –If shape of *input* is not the same as *target*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inputs = Tensor(np.array([[0.1, 0.2, 0.4, 0.8], [0.2, 0.3, 0.5, 0.7]]), mindspore.
↪float32)
>>> target = Tensor(np.array([[1, 2, 0, 3], [2, 3, -1, 1]]), mindspore.int32)
>>> output = ops.multilabel_margin_loss(inputs, target)
>>> print(output)
0.325
```

mindspore.ops.multilabel_soft_margin_loss

mindspore.ops.**multilabel_soft_margin_loss**(*input*, *target*, *weight=None*, *reduction='mean'*)

Calculates the MultiLabelSoftMarginLoss. The multi-label soft margin loss is a commonly used loss function in multi-label classification tasks where an input sample can belong to multiple classes. Given an input $input$ and binary labels $output$ of size (N, C), where N denotes the number of samples and C denotes the number of classes.

$$loss\left(input, output\right) = -\frac{1}{N}\frac{1}{C}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(output_{ij}\log\frac{1}{1+e^{-input_{ij}}} + \left(1-output_{ij}\right)\log\frac{e^{-input_{ij}}}{1+e^{-input_{ij}}}\right)$$

where $input_{ij}$ represents the predicted score of sample i for class j. $output_{ij}$ represents the binary label of sample i for class j, where sample i belongs to class j if $output_{ij} = 1$, and sample i does not belong to class j if $output_{ij} = 0$. For a multi-label classification task, each sample may have multiple labels with a value of 1 in the binary label $output$. *weight* will multiply to the loss of each class if given.

Parameters

- **input** (Tensor) –A tensor of shape (N, C) , where N is batch size and C is number of classes.
- **target** (Tensor) –The label target Tensor which has the same shape as *input*.
- **weight** (*Union[Tensor, int, float], optional*) –The manual rescaling weight given to each class. Default: `None`.

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor, the data type is the same as input, if the *reduction* is `'none'`, its shape is (N) , otherwise it is zero.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> input = Tensor([[0.3, 0.6, 0.6], [0.9, 0.4, 0.2]])
>>> target = Tensor([[0.0, 0.0, 1.0], [0.0, 0.0, 1.0]])
>>> loss = ops.multilabel_soft_margin_loss(input, target, reduction='mean')
>>> print(loss.asnumpy())
0.84693956
```

mindspore.ops.nll_loss

`mindspore.ops.`**`nll_loss`**(*inputs*, *target*, *weight=None*, *ignore_index=- 100*, *reduction='mean'*, *label_smoothing=0.0*)

Gets the negative log likelihood loss between inputs and target.

The nll loss with reduction=none can be described as:

$$\ell(x,t) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_{t_n} x_{n,t_n}, \quad w_c = \text{weight}\,[c] \cdot \mathbb{1}\{c \neq \text{ignore\_index}\},$$

where x is the inputs, t is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `'None'` (default `'mean'`), then

$$\ell(x,t) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{tn}} l_n, & \text{if reduction} = \text{'mean'}, \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'} \end{cases}$$

Parameters

- **inputs** (`Tensor`) –(N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$. *inputs* is expected to be log-probabilities, data type must be float16 or float32.
- **target** (`Tensor`) –(N) or $(N, d_1, d_2, ..., d_K)$ for high-dimensional loss, data type must be int32.
- **weight** (`Tensor`) –A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$. The data type must be float16 or float32. Default: `None` .
- **ignore_index** (*int*) –Specifies a target value that is ignored and does not contribute to the input gradient. Default: `-100` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **label_smoothing** (*float*) –Label smoothing values, a regularization tool used to prevent the model from overfitting when calculating Loss. The value range is [0.0, 1.0]. Default value: `0.0`.

Returns

Tensor, the computed loss value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inputs = mindspore.Tensor(np.random.randn(3, 5), mindspore.float32)
>>> target = mindspore.Tensor(np.array([1, 0, 4]), mindspore.int32)
>>> output = ops.nll_loss(inputs, target)
```

mindspore.ops.smooth_l1_loss

`mindspore.ops.`**smooth_l1_loss**(*input*, *target*, *beta=1.0*, *reduction='none'*)

Computes smooth L1 loss, a robust L1 loss.

SmoothL1Loss is a Loss similar to MSELoss but less sensitive to outliers as described in the Fast R-CNN by Ross Girshick.

Given two input x, y of length N, the unreduced SmoothL1Loss can be described as follows:

$$L_i = \begin{cases} \frac{0.5(x_i - y_i)^2}{\text{beta}}, & \text{if } |x_i - y_i| < \text{beta} \\ |x_i - y_i| - 0.5 * \text{beta}, & \text{otherwise.} \end{cases}$$

If *reduction* is not *none*, then:

$$L = \begin{cases} \text{mean}(L_i), & \text{if reduction = 'mean';} \\ \text{sum}(L_i), & \text{if reduction = 'sum'.} \end{cases}$$

Here beta controls the point where the loss function changes from quadratic to linear. beta > 0, its default value is `1.0`. N is the batch size.

> **Warning:** This API has poor performance on CPU and it is recommended to run it on the Ascend/GPU.

Parameters

- **input** (`Tensor`) –Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32, float64.
- **target** (`Tensor`) –Ground truth data, tensor of shape $(N, *)$.

  - CPU/Ascend: has the same shape as the *input*, *target* and *input* comply with the implicit type conversion rules to make the data types consistent.
  - GPU: has the same shape and dtype as the *input*.
- **beta** (*number, optional*) – A parameter used to control the point where the function will change between L1 to L2 loss. Default: `1.0`.
  - Ascend: The value should be equal to or greater than zero.
  - CPU/GPU: The value should be greater than zero.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'none'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor, if *reduction* is `'none'`, then output is a tensor with the same shape as *input*. Otherwise, the shape of output tensor is ().

Raises

- **TypeError** – If input *input*, *target* is not Tensor.
- **RuntimeError** – If dtype of *input* or *target* is not one of float16, float32, float64, bfloat16.
- **ValueError** – If shape of *input* is not the same as *target*.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **TypeError** – If *beta* is not a float, int or bool.
- **RuntimeError** – If *beta* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = ops.smooth_l1_loss(logits, labels)
>>> print(output)
[0.  0.  0.5]
```

mindspore.ops.soft_margin_loss

mindspore.ops.**soft_margin_loss**(*input*, *target*, *reduction='mean'*)

Calculate the soft margin loss of input and target.

Creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1).

$$\text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{\text{x.nelement()}}$$

where $x.nelement()$ is the number of elements of x.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −Predict data. Data type must be float16, float32, bfloat16 (Atlas training series products are not supported).
- **target** (Tensor) −Ground truth data, with the same shape as *input*. In GE mode, the data type should be the same as *input*.
- **reduction** (*str, optional*) −Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar. If *reduction* is `'none'`, its shape is the same as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** −If *input* or *target* is not a Tensor.
- **TypeError** −If dtype of *input* or *target* is not float16, float32, bfloat16 (Atlas training series products are not supported).
- **ValueError** −If shape of *input* is not the same as that of *target*.
- **ValueError** −If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([[0.3, 0.7], [0.5, 0.5]]), mindspore.float32)
>>> labels = Tensor(np.array([[-1, 1], [1, -1]]), mindspore.float32)
>>> output = ops.soft_margin_loss(logits, labels)
>>> print(output)
0.6764238
```

mindspore.ops.triplet_margin_loss

mindspore.ops.**triplet_margin_loss**(*anchor*, *positive*, *negative*, *margin=1.0*, *p=2*, *eps=1e-06*, *swap=False*, *reduction='mean'*)

TripletMarginLoss operation. See *mindspore.nn.TripletMarginLoss* for details.

Parameters

- **anchor** (Tensor) – A sample randomly selected from the training set. Data type must be BasicType.
- **positive** (Tensor) – A sample belonging to the same category as *anchor*, with the same type and shape as *anchor*.
- **negative** (Tensor) – A sample belonging to the different class from *anchor*, with the same type and shape as *anchor*.
- **margin** (*float, optional*) – Make a margin between the positive pair and the negative pair. The shape of margin must be 0. Default: 1.0 .
- **p** (*int, optional*) – The degree of norm for pairwise distance. Default: 2 .
- **eps** (*float, optional*) – Add small value to avoid division by zero. Default: 1e-06.
- **swap** (*bool, optional*) – The distance swap change the negative distance to the distance between positive sample and negative sample. Default: False .
- **reduction** (*str, optional*) – Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the mean of elements in the output.
  - 'sum': the output elements will be summed.

Returns

Tensor. If *reduction* is "none", its shape is (N). Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If *anchor* or *positive* or *negative* is not a Tensor.
- **TypeError** – If dtype of *anchor*, *positive* and *negative* is not the same.
- **TypeError** – If *margin* is not a float.
- **TypeError** – If *p* is not an int.
- **TypeError** – If *eps* is not a float.
- **TypeError** – If *swap* is not a bool.
- **ValueError** – If dimensions of input *anchor*, *positive* and *negative* are less than or equal to 1 at the same time.

- **ValueError** – If the dimension of input *anchor* or *positive* or *negative* is bigger than or equal to 8.
- **ValueError** – If shape of *anchor*, *positive* and *negative* cannot broadcast.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> anchor = Tensor(np.array([[0.3, 0.7], [0.5, 0.5]]), mindspore.float32)
>>> positive = Tensor(np.array([[0.4, 0.6], [0.4, 0.6]]), mindspore.float32)
>>> negative = Tensor(np.array([[0.2, 0.9], [0.3, 0.7]]), mindspore.float32)
>>> output = ops.triplet_margin_loss(anchor, positive, negative)
>>> print(output)
0.8881968
```

2.3.3 Activation Functions

API Name	Description	Supported Platforms
mindspore.ops.celu	celu activation function, computes celu (Continuously differentiable exponential linear units) of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.elu	Exponential Linear Unit activation function.	Ascend GPU CPU
mindspore.ops.fast_gelu	Fast Gaussian Error Linear Units activation function.	Ascend GPU CPU
mindspore.ops.gelu	Gaussian Error Linear Units activation function.	Ascend GPU CPU
mindspore.ops.glu	Computes GLU (Gated Linear Unit activation function) of input tensors.	Ascend GPU CPU
mindspore.ops.gumbel_softmax	Returns the samples from the Gumbel-Softmax distribution and optionally discretizes.	Ascend GPU CPU
mindspore.ops.hardshrink	Hard Shrink activation function.	Ascend GPU CPU
mindspore.ops.hardsigmoid	Hard Sigmoid activation function.	Ascend GPU CPU
mindspore.ops.hardswish	Hard Swish activation function.	Ascend GPU CPU
mindspore.ops.hardtanh	Applies the hardtanh activation function element-wise.	Ascend GPU CPU
mindspore.ops.leaky_relu	leaky_relu activation function.	Ascend GPU CPU
mindspore.ops.log_softmax	Applies the Log Softmax function to the input tensor on the specified axis.	Ascend GPU CPU
mindspore.ops.logsigmoid	Applies LogSigmoid activation element-wise.	Ascend GPU CPU
mindspore.ops.mish	Computes MISH(A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.prelu	Parametric Rectified Linear Unit activation function.	Ascend GPU CPU

continues on next page

Table 13 – continued from previous page

mindspore.ops.relu	Computes ReLU (Rectified Linear Unit activation function) of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.relu6	Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.rrelu	Randomized Leaky ReLU activation function.	Ascend GPU CPU
mindspore.ops.selu	Activation function SeLU (Scaled exponential Linear Unit).	Ascend GPU CPU
mindspore.ops.sigmoid	Computes Sigmoid of input element-wise.	Ascend GPU CPU
mindspore.ops.silu	Computes Sigmoid Linear Unit of input element-wise, also known as Swish function.	Ascend GPU CPU
mindspore.ops.softmax	Applies the Softmax operation to the input tensor on the specified axis.	Ascend GPU CPU
mindspore.ops.softmin	Applies the Softmin operation to the input tensor on the specified axis.	Ascend GPU CPU
mindspore.ops.softshrink	Soft Shrink activation function.	Ascend GPU CPU
mindspore.ops.softsign	SoftSign activation function.	Ascend GPU CPU
mindspore.ops.swiglu	Computes SwiGLU (Swish-Gated Linear Unit activation function) of input tensor.	Ascend
mindspore.ops.tanh	Computes hyperbolic tangent of input element-wise.	Ascend GPU CPU
mindspore.ops.threshold	Returns each element of *input* after thresholding by *thr* as a Tensor.	Ascend GPU CPU

mindspore.ops.celu

mindspore.ops.**celu**(*x*, *alpha=1.0*)

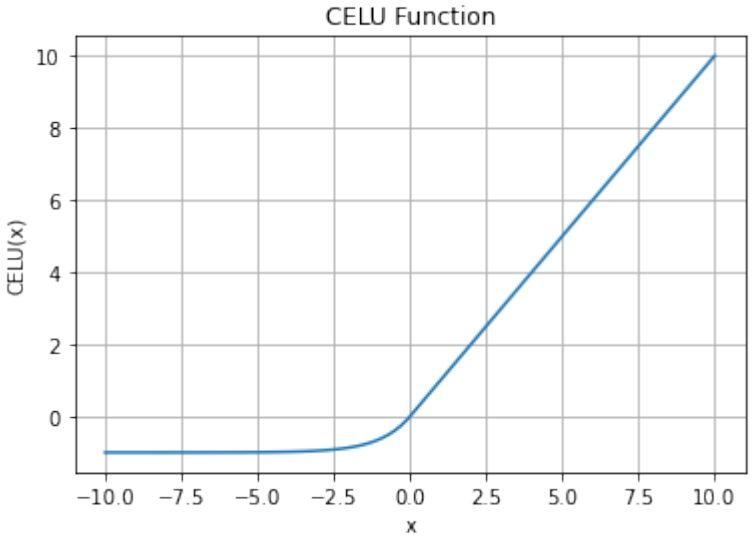
celu activation function, computes celu (Continuously differentiable exponential linear units) of input tensors element-wise. The formula is defined as follows:

$$\text{CeLU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

For more details, please refer to celu.

Warning: This is an experimental API that is subject to change or deletion.

CELU Activation Function Graph:



Parameters

- **x** (Tensor) −The input of celu with data type of float16 or float32.
- **alpha** (*float, optional*) −The α value for the Celu formulation. Default: 1.0

Returns

Tensor, has the same data type and shape as the input.

Raises

- **TypeError** −If *alpha* is not a float.
- **TypeError** −If *x* is not a Tensor.
- **TypeError** −If dtype of *x* is neither float16 nor float32.
- **ValueError** −If *alpha* has the value of 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([-2.0, -1.0, 1.0, 2.0]), mindspore.float32)
>>> output = ops.celu(x, alpha=1.0)
>>> print(output)
[-0.86466473 -0.63212055  1.          2.        ]
```

mindspore.ops.elu

mindspore.ops.**elu**(*input_x*, *alpha=1.0*)

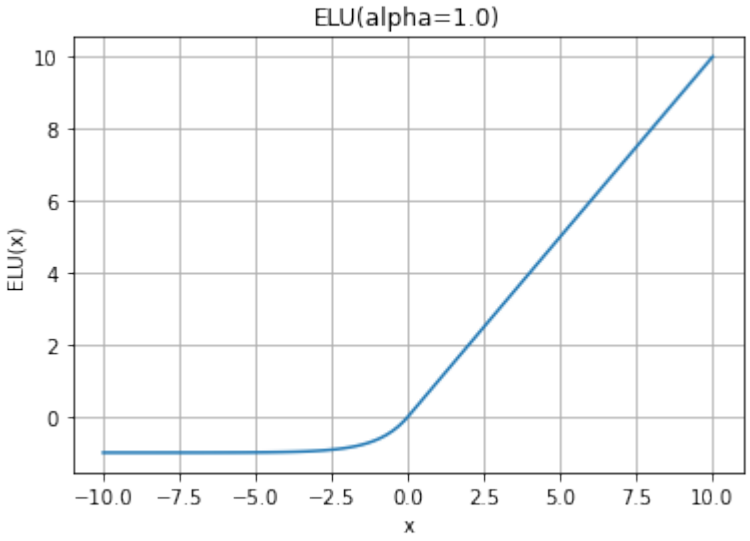
Exponential Linear Unit activation function.

Applies the exponential linear unit function element-wise. The activation function is defined as:

$$\text{ELU}(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

Where x is the element of input Tensor *input_x*, α is param *alpha*, it determines the smoothness of ELU.

ELU function graph:



Parameters

- **input_x** (Tensor) −The input of ELU is a Tensor of any dimension with data type of float16 or float32.
- **alpha** (*float, optional*) −The alpha value of ELU, the data type is float. Only support `1.0` currently. Default: `1.0`.

Returns

Tensor, has the same shape and data type as *input_x*.

Raises

- **TypeError** −If *alpha* is not a float.
- **TypeError** −If dtype of *input_x* is neither float16 nor float32.
- **ValueError** −If *alpha* is not equal to 1.0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = ops.elu(x)
>>> print(output)
[[-0.63212055  4.         -0.99966455]
 [ 2.         -0.99326205  9.        ]]
```

mindspore.ops.fast_gelu

mindspore.ops.**fast_gelu**(*x*)

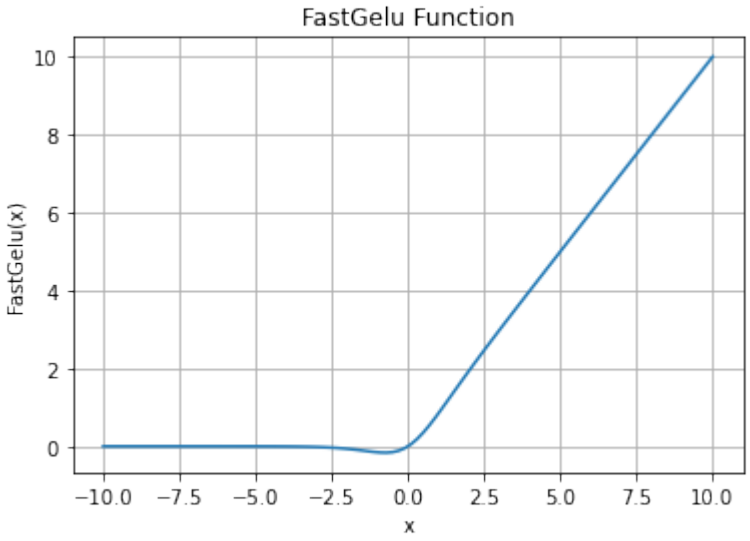
Fast Gaussian Error Linear Units activation function.

FastGeLU is defined as follows:

$$\text{output} = \frac{x}{1 + \exp(-1.702 * |x|)} * \exp(0.851 * (x - |x|)),$$

where x is the element of the input.

FastGelu function graph:



Parameters

x (Tensor) – Input to compute the FastGeLU with data type of float16 or float32.

Returns

Tensor, with the same type and shape as x.

Raises

TypeError – If dtype of x is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = ops.fast_gelu(x)
>>> print(output)
[[-1.5418735e-01  3.9921875e+00 -9.7473649e-06]
 [ 1.9375000e+00 -1.0052517e-03  8.9824219e+00]]
```

mindspore.ops.gelu

mindspore.ops.**gelu**(*input*, *approximate='none'*)

Gaussian Error Linear Units activation function.

GeLU is described in the paper Gaussian Error Linear Units (GELUs). And also please refer to BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

When *approximate* argument is *none*, GeLU is defined as follows:

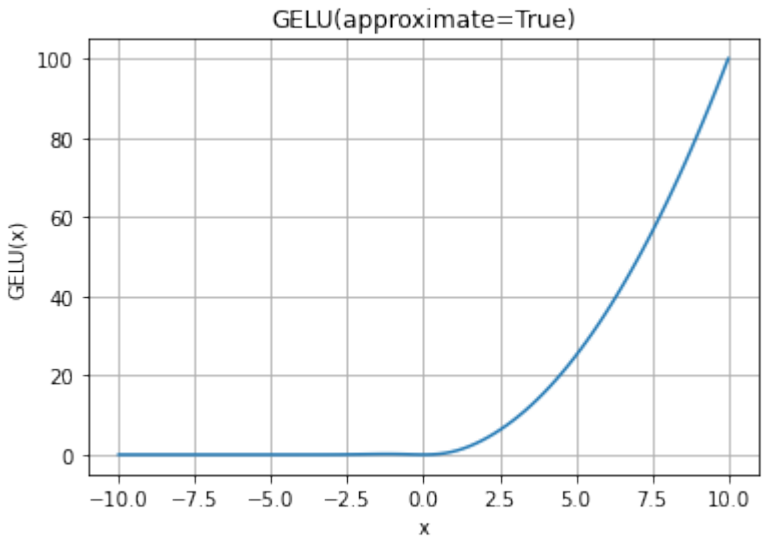
$$GELU(x_i) = x_i * P(X < x_i),$$

where P is the cumulative distribution function of the standard Gaussian distribution, x_i is the input element.

When *approximate* argument is *tanh*, GeLU is estimated with:

$$GELU(x_i) = 0.5 * x_i * (1 + \tanh(\sqrt{(2/\pi)} * (x_i + 0.044715 * x_i^3)))$$

GELU Activation Function Graph:



Parameters

- **input** (Tensor) –The input of the activation function GeLU, the data type is float16, float32 or float64.
- **approximate** (*str, optional*) –the gelu approximation algorithm to use. Acceptable vaslues are 'none' and 'tanh'. Default: 'none'.

Returns

Tensor, with the same type and shape as *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If dtype of *input* is not bfloat16, float16, float32 or float64.
- **ValueError** – If *approximate* value is neither *none* nor *tanh*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> result = ops.gelu(x, approximate='none')
>>> print(result)
[0.8413447 1.9544997 2.9959505]
```

mindspore.ops.glu

`mindspore.ops.`**`glu`**(*x*, *axis=- 1*)

Computes GLU (Gated Linear Unit activation function) of input tensors.

$$GLU(a, b) = a \otimes \sigma(b)$$

where a is the first half of the input matrices and b is the second half.

Here σ is the sigmoid function, and $\otimes$ is the Hadamard product. See Language Modeling with Gated Convluational Networks.

Parameters

- **x** (`Tensor`) – Tensor to be split. Its dtype is Number, and shape is $(*_1, N, *_2)$ where * means, any number of additional dimensions.
- **axis** (*int, optional*) – the axis to split the input. It must be int. Default: `-1` , the last axis of x.

Returns

Tensor, the same dtype as the x, with the shape $(*_1, M, *_2)$ where $M = N/2$.

Raises

- **TypeError** – If dtype of x is not Number.
- **TypeError** – If x is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> input = Tensor([[0.1,0.2,0.3,0.4],[0.5,0.6,0.7,0.8]])
>>> output = ops.glu(input)
>>> print(output)
[[0.05744425 0.11973753]
 [0.33409387 0.41398472]]
```

mindspore.ops.gumbel_softmax

mindspore.ops.**gumbel_softmax**(*logits*, *tau=1.0*, *hard=False*, *dim=- 1*)

Returns the samples from the Gumbel-Softmax distribution and optionally discretizes. If *hard* is `True`, the returned samples will be one-hot, otherwise it will be probability distributions that sum to 1 across *dim*.

Parameters

- **logits** (Tensor) –Unnormalized log probabilities. The data type must be float16 or float32.
- **tau** (*float, optional*) –The scalar temperature, which is a positive number. Default: `1.0` .
- **hard** (*bool, optional*) –If `True`, the returned samples will be discretized as one-hot vectors, but will be differentiated as if it is the soft sample in autograd. Default: `False` .
- **dim** (*int, optional*) –Dim for softmax to compute. Default: `-1` .

Returns

Tensor, has the same dtype and shape as *logits*.

Raises

- **TypeError** –If *logits* is not a Tensor.
- **TypeError** –If dtype of *logits* is not one of: float16, float32.
- **TypeError** –If *tau* is not a float.
- **TypeError** –If *hard* is not a bool.
- **TypeError** –If *dim* is not an int.
- **ValueError** –If If *tau* is not positive.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> output = ops.gumbel_softmax(input_x, 1.0, True, -1)
>>> print(output.shape)
(2, 3)
```

mindspore.ops.hardshrink

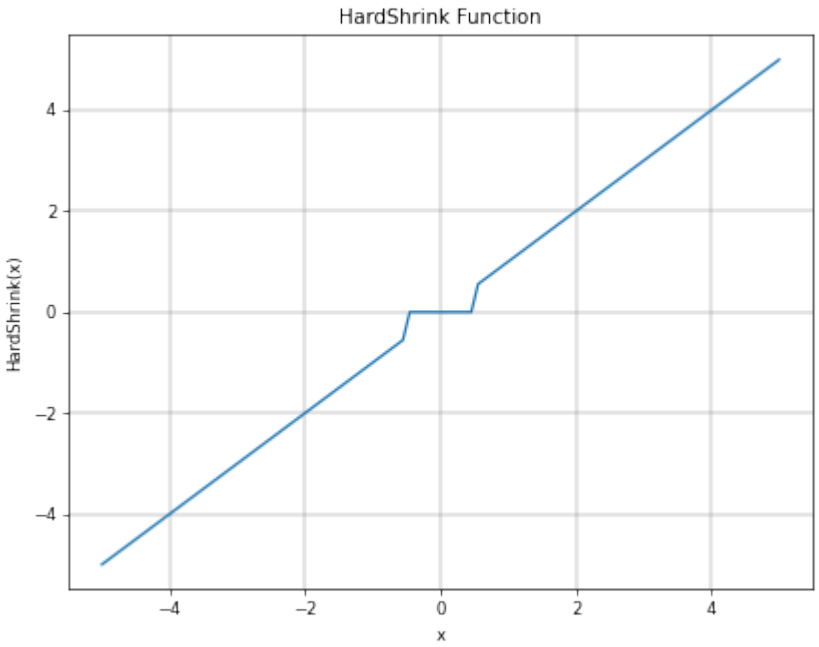
mindspore.ops.**hardshrink**(*input*, *lambd=0.5*)

Hard Shrink activation function. Calculates the output according to the input elements.

The formula is defined as follows:

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

HardShrink Activation Function Graph:



Parameters

- **input** (Tensor) – The input of Hard Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32.
- **lambd** (*number, optional*) – The threshold λ defined by the Hard Shrink formula. Default: 0.5 .

Returns

Tensor, has the same data type and shape as the input *input*.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([[0.5, 1, 2.0], [0.0533, 0.0776, -2.1233]]), mindspore.float32)
>>> output = ops.hardshrink(input)
>>> print(output)
[[ 0.      1.      2.    ]
 [ 0.      0.     -2.1233]]
```

mindspore.ops.hardsigmoid

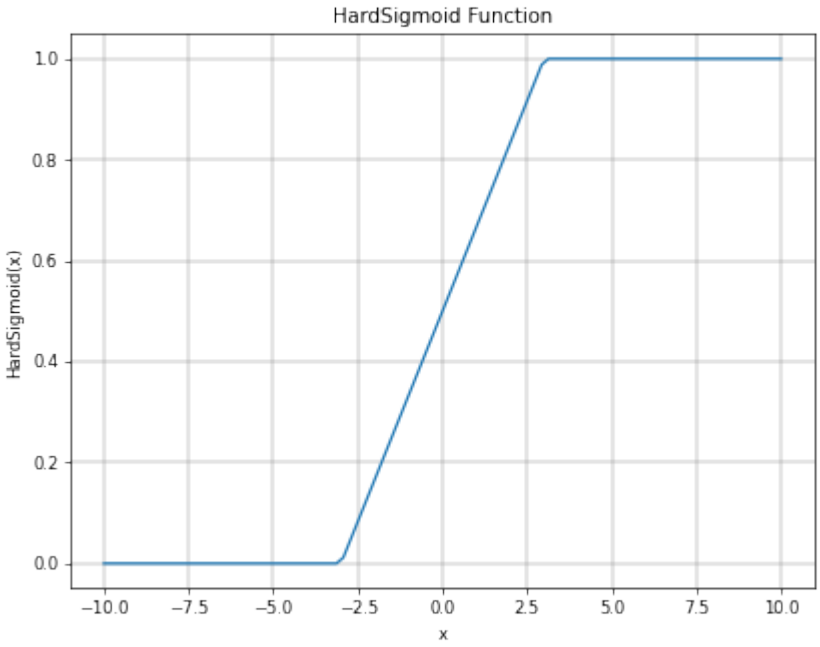
mindspore.ops.**hardsigmoid**(*input*)

Hard Sigmoid activation function. Calculates the output according to the input elements.

Hard Sigmoid is defined as:

$$\text{HardSigmoid}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ 1, & \text{if } input \geq +3, \\ input/6 + 1/2, & \text{otherwise} \end{cases}$$

HardSigmoid Activation Function Graph:



Parameters

input (Tensor) – The input Tensor.

Returns

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> output = ops.hardsigmoid(input)
>>> print(output)
[0.3333 0.1666 0.5    0.8335 0.6665]
```

mindspore.ops.hardswish

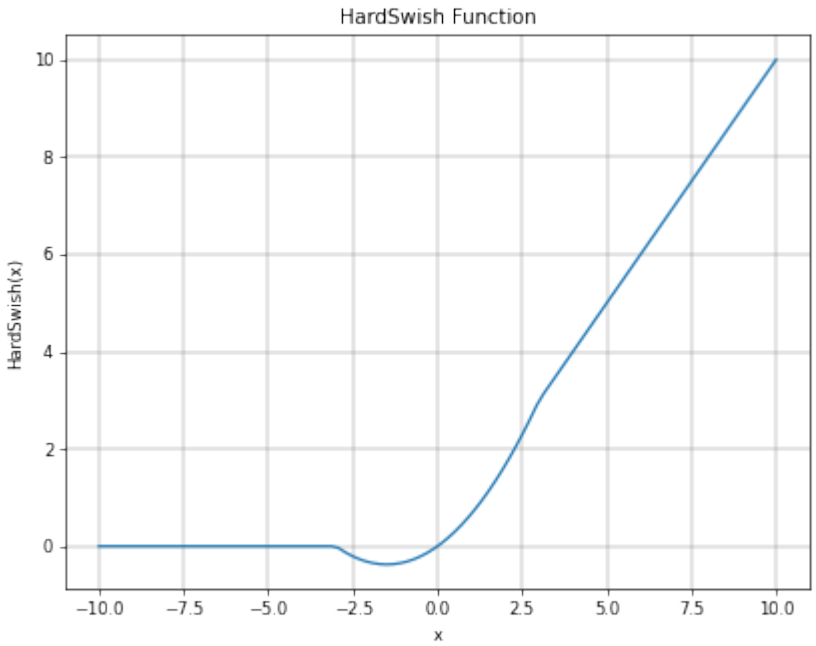
`mindspore.ops.`**`hardswish`**(*input*)

Hard Swish activation function. The input is a Tensor with any valid shape.

Hard swish is defined as:

$$\text{HardSwish}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ input, & \text{if } input \geq +3, \\ input * (input + 3)/6, & \text{otherwise} \end{cases}$$

HardSwish Activation Function Graph:



Parameters

input (Tensor) –The input Tensor.

Returns

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *input* is neither int nor float.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> output = ops.hardswish(input)
>>> print(output)
[-0.3333  -0.3333  0  1.667  0.6665]
```

mindspore.ops.hardtanh

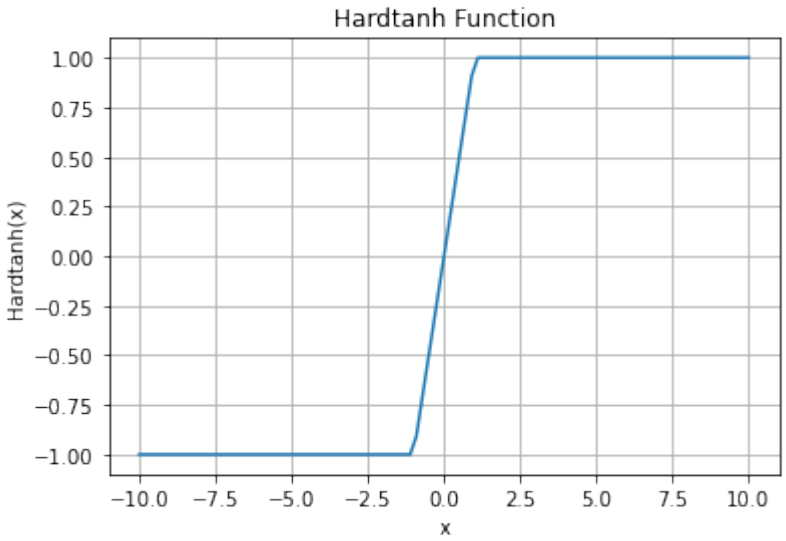
mindspore.ops.**hardtanh**(*input*, *min_val=- 1.0*, *max_val=1.0*)

Applies the hardtanh activation function element-wise. The activation function is defined as:

$$\text{hardtanh}(input) = \begin{cases} max\_val, & \text{if } input > max\_val \\ min\_val, & \text{if } input < min\_val \\ input, & \text{otherwise.} \end{cases}$$

Linear region range $[min\_val, max\_val]$ can be adjusted using *min_val* and *max_val*.

Hardtanh Activation Function Graph:



Parameters

- **input** (Tensor) – Input Tensor.
- **min_val** (*Union[int, float], optional*) – Minimum value of the linear region range. Default: `-1.0`.
- **max_val** (*Union[int, float], optional*) – Maximum value of the linear region range. Default: `1.0`.

Returns

Tensor, with the same dtype and shape as *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If dtype of *min_val* is neither float nor int.
- **TypeError** – If dtype of *max_val* is neither float nor int.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([-1, -2, 0, 2, 1], mindspore.float16)
>>> output = ops.hardtanh(x, min_val=-1.0, max_val=1.0)
>>> print(output)
[-1. -1.  0.  1.  1.]
```

mindspore.ops.leaky_relu

mindspore.ops.**leaky_relu**(*input*, *alpha=0.2*)

leaky_relu activation function. The element of *input* less than 0 times *alpha* .

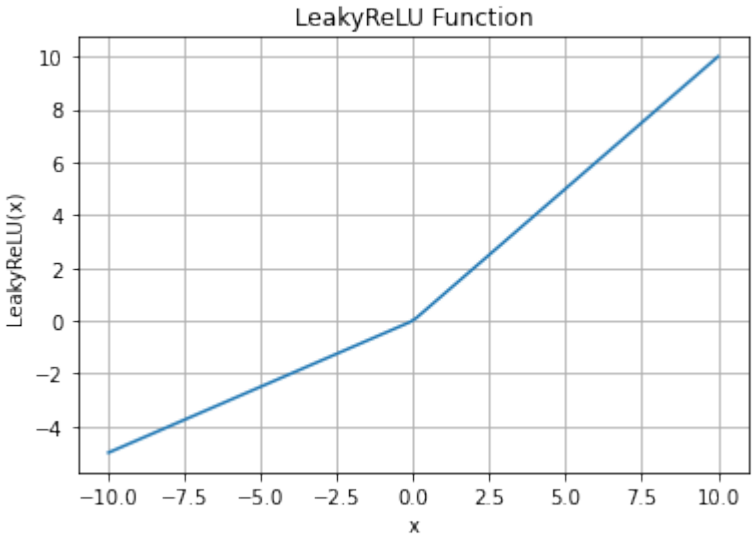
The activation function is defined as:

$$\text{leaky\_relu}(input) = \begin{cases} input, & \text{if } input \geq 0; \\ \alpha * input, & \text{otherwise.} \end{cases}$$

where α represents the *alpha* parameter.

For more details, see Rectifier Nonlinearities Improve Neural Network Acoustic Models.

LeakyReLU Activation Function Graph:



Parameters

- **input** (Tensor) – The input of leaky_relu is a Tensor of any dimension.
- **alpha** (*Union[int, float], optional*) – Slope of the activation function when the element of *input* is less than 0. Default: 0.2 .

Returns

Tensor, has the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *alpha* is not a float or an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> print(ops.leaky_relu(x, alpha=0.2))
[[-0.2  4.  -1.6]
 [ 2.  -1.   9. ]]
```

mindspore.ops.log_softmax

mindspore.ops.**log_softmax**(*logits*, *axis=- 1*)

Applies the Log Softmax function to the input tensor on the specified axis. Supposes a slice in the given axis, x for each element x_i, the Log Softmax function is shown as follows:

$$\text{output}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)}\right),$$

where N is the length of the Tensor.

Parameters

- **logits** (Tensor) –The input Tensor, which is the x in the formula above, it's shape is $(N, *)$, where $*$ means, any number of additional dimensions, with float16 or float32 data type.
- **axis** (*int*) –The axis to perform the Log softmax operation. Default: `-1` .

Returns

Tensor, with the same type and shape as the logits.

Raises

- **TypeError** –If *axis* is not an int.
- **TypeError** –If dtype of *logits* is neither float16 nor float32.
- **ValueError** –If *axis* is not in range [-len(logits.shape), len(logits.shape)).
- **ValueError** –If dimension of *logits* is less than 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = ops.log_softmax(logits)
>>> print(output)
[-4.4519143 -3.4519143 -2.4519143 -1.4519144 -0.4519144]
```

mindspore.ops.logsigmoid

mindspore.ops.**logsigmoid**(*x*)

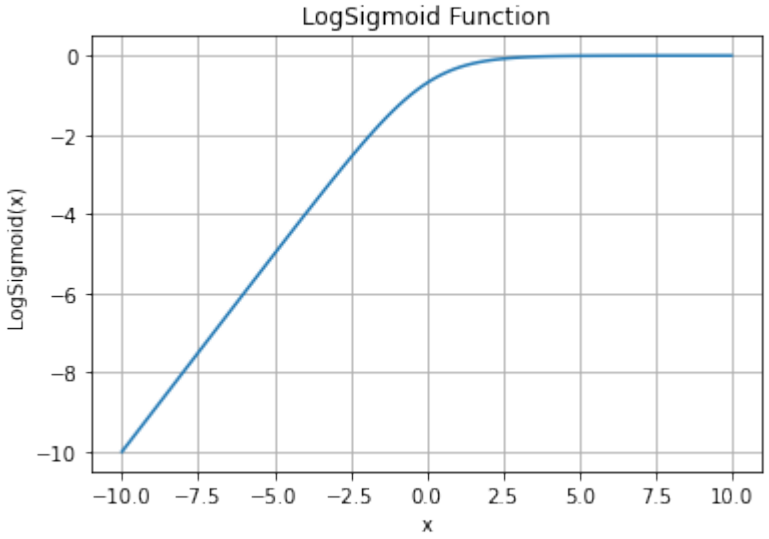
Applies LogSigmoid activation element-wise. The input is a Tensor with any valid shape.

Logsigmoid is defined as:

$$\text{logsigmoid}(x_i) = \log(\frac{1}{1 + \exp(-x_i)}),$$

where x_i is the element of the input.

LogSigmoid Activation Function Graph:



Parameters

x (Tensor) −The input of LogSigmoid with data type of float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Returns

Tensor, with the same type and shape as the *x*.

Raises

TypeError −If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> output = ops.logsigmoid(x)
>>> print(output)
[-0.31326166 -0.12692806 -0.04858734]
```

mindspore.ops.mish

mindspore.ops.**mish**(*x*)

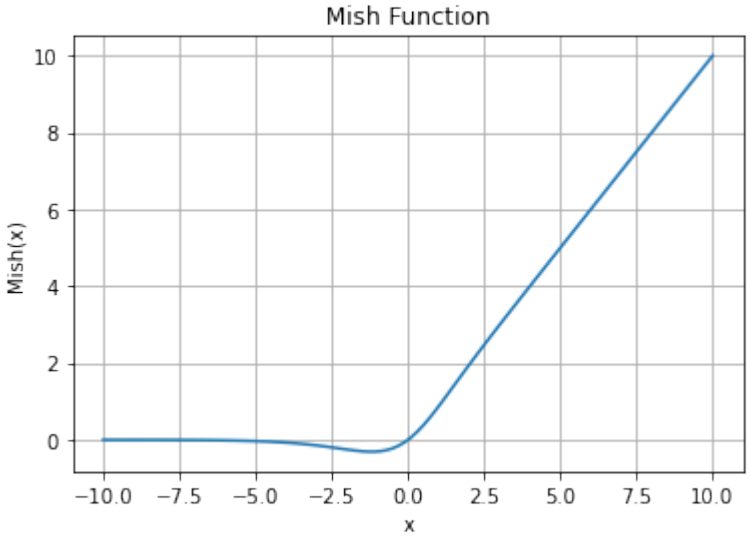
Computes MISH(A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.

The function is shown as follows:

$$\text{output} = x * \tanh(\log(1 + \exp(x)))$$

See more details in A Self Regularized Non-Monotonic Neural Activation Function.

Mish Activation Function Graph:



Parameters

x (Tensor) −The input Tensor. Supported dtypes:

- GPU/CPU: float16, float32, float64.
- Ascend: float16, float32.

Returns

Tensor, with the same type and shape as the *x*.

Raises

TypeError −If dtype of *x* is not float16, float32 or float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = ops.mish(input_x)
>>> print(output)
[[-3.0340147e-01  3.9974129e+00 -2.68311895e-03]
 [ 1.9439590e+00  -3.3576239e-02 8.99999990e+00]]
>>> input_x = Tensor(2.1, mindspore.float32)
>>> output = ops.mish(input_x)
>>> print(output)
2.050599
```

mindspore.ops.prelu

mindspore.ops.**prelu**(*input*, *weight*)

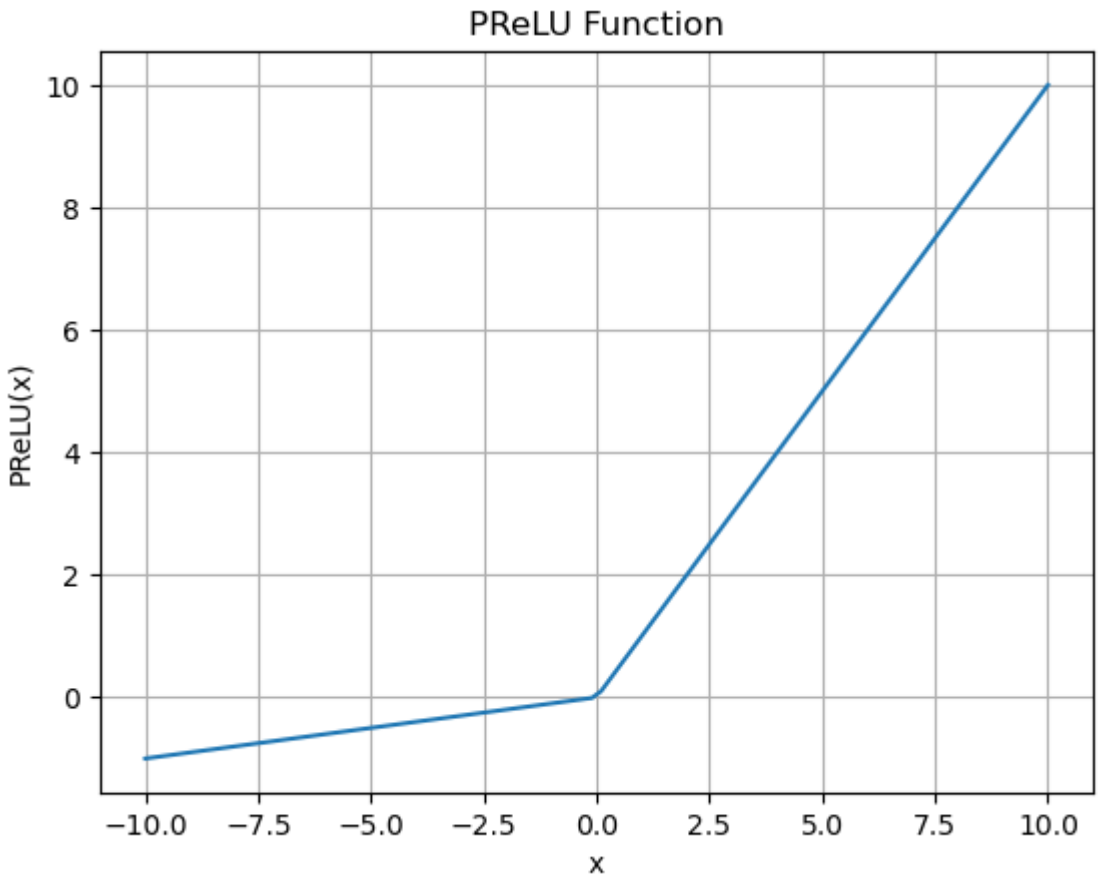
Parametric Rectified Linear Unit activation function.

PReLU is described in the paper Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Defined as follows:

$$prelu(x_i) = \max(0, x_i) + \min(0, w * x_i),$$

where x_i is an element of a channel of the input, w is the weight of the channel.

PReLU Activation Function Graph:



Note: Channel dim is the 2nd dim of input. When input has dims < 2, then there is no channel dim and the number of channels = 1.

Parameters

- **input** (Tensor) −The input Tensor of the activation function.
- **weight** (Tensor) −Weight Tensor. The size of the weight should be 1 or the number of channels at Tensor *input*.

Returns

Tensor, with the same shape and dtype as *input*. For detailed information, please refer to *mindspore.mint.nn.PReLU*.

Raises

- **TypeError** −If the *input* or the *weight* is not a Tensor.
- **ValueError** −If the *weight* is not a 0-D or 1-D Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(-6, 6).reshape((2, 3, 2)), mindspore.float32)
>>> weight = Tensor(np.array([0.1, 0.6, -0.3]), mindspore.float32)
>>> output = ops.prelu(x, weight)
>>> print(output)
[[[-0.60 -0.50]
  [-2.40 -1.80]
  [ 0.60  0.30]]
 [[ 0.00  1.00]
  [ 2.00  3.00]
  [ 4.0   5.00]]]
```

mindspore.ops.relu

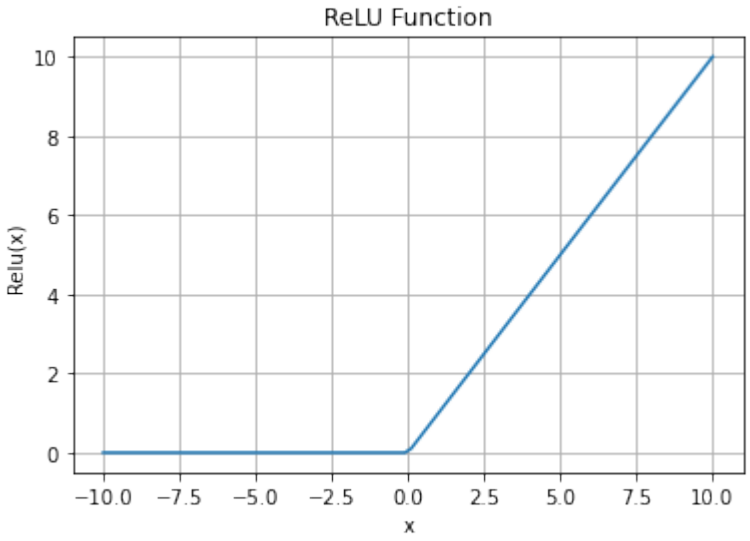
mindspore.ops.**relu**(*input*, *inplace=False*)

Computes ReLU (Rectified Linear Unit activation function) of input tensors element-wise.

It returns $\max(input,\ 0)$ element-wise. Specially, the neurons with the negative output will be suppressed and the active neurons will stay the same.

$$ReLU(input) = (input)^+ = \max(0, input)$$

ReLU Activation Function Graph:



Parameters

- **input** (Tensor) −The input Tensor.
- **inplace** (*bool, optional*) −Whether to use inplace mode, Defaults to False.

Returns

Tensor, with the same dtype and shape as the *input*.

Raises

- **TypeError** –If dtype of *input* is not Number type.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = ops.relu(input)
>>> print(output)
[[0. 4. 0.]
 [2. 0. 9.]]
```

mindspore.ops.relu6

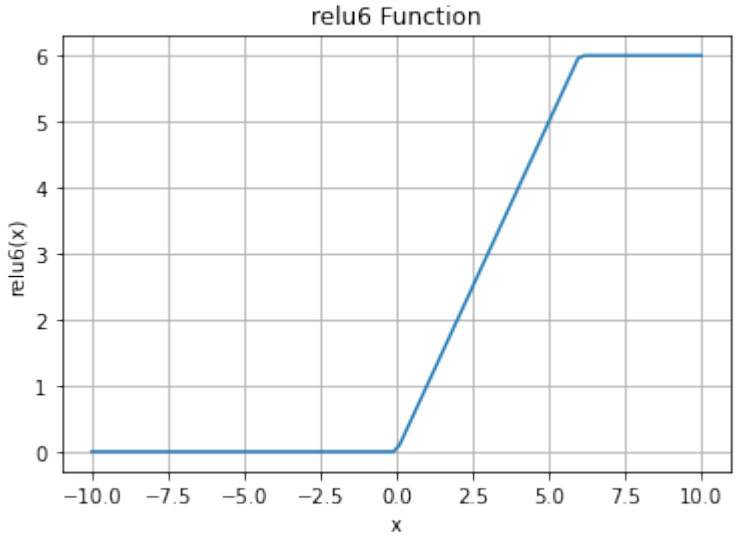
`mindspore.ops.`**`relu6`**`(x)`

Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input tensors element-wise.

$$\text{ReLU6}(x) = \min(\max(0, x), 6)$$

It returns $\min(\max(0, x), 6)$ element-wise.

ReLU6 Activation Function Graph:



Parameters

x (`Tensor`) –Tensor of shape $(N, *)$, where $*$ means any number of additional dimensions. Data type must be float16, float32.

Returns

Tensor, with the same dtype and shape as the *x*.

Raises

- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **TypeError** – If *x* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> result = ops.relu6(x)
>>> print(result)
[[0. 4. 0.]
 [2. 0. 6.]]
```

mindspore.ops.rrelu

`mindspore.ops.`**`rrelu`**(*input*, *lower=1.0 / 8*, *upper=1.0 / 3*)

Randomized Leaky ReLU activation function.

The activation function is defined as:

$$\text{rrelu}(input_{ji}) = \begin{cases} input_{ji}, & \text{if } input_{ji} \geq 0; \\ \alpha_{ji} * input_{ji}, & \text{otherwise.} \end{cases}$$

where $\alpha_{ji} \sim U(l, u), l \leq u$.

Applies the rrelu function elementally, as described in the paper: Empirical Evaluation of Rectified Activations in Convolution Network .

Parameters

- **input** (`Tensor`) – The input of rrelu is a Tensor of any dimension.
- **lower** (*`Union[int, float]`*) – Slope of the activation function at data of *input* is less than 0. Default: `1.0 / 8`.
- **upper** (*`Union[int, float]`*) – Slope of the activation function at data of *input* is less than 0. Default: `1.0 / 3`.

Returns

Tensor, after rrelu, has the same type and shape as the *input*.

Raises

- **TypeError** – If *lower* is not a float or an int.
- **TypeError** – If *upper* is not a float or an int.
- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *input* is not a Tensor of mindspore.float16 or mindspore.float32.
- **ValueError** – If *lower* is greater than upper.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0], [2.0, 0]]), mindspore.float32)
>>> output = ops.rrelu(x)
>>> print(output)
[[-0.31465699  4.        ]
 [ 2.          0.        ]]
```

mindspore.ops.selu

`mindspore.ops.`**`selu`**(*input_x*)

Activation function SeLU (Scaled exponential Linear Unit).

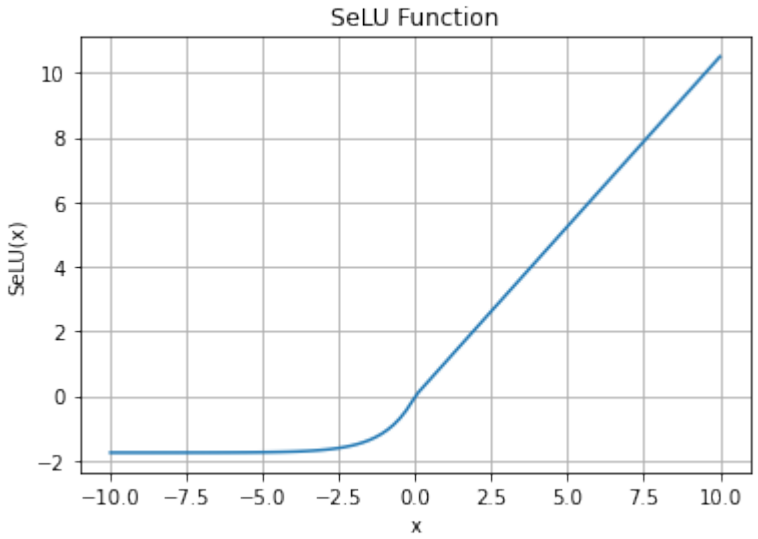
The activation function is defined as:

$$E_i = scale * \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \text{alpha} * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where $alpha$ and $scale$ are pre-defined constants($alpha = 1.67326324$ and $scale = 1.05070098$).

See more details in Self-Normalizing Neural Networks.

SeLU Activation Function Graph:



Parameters

`input_x` (`Tensor`) −Tensor of any dimension, the data type is int8, int32, float16, float32, or float64 (CPU, GPU only).

Returns

Tensor, with the same type and shape as the *input_x*.

Raises

TypeError – If dtype of *input_x* is not int8, int32, float16, float32, or float64.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = ops.selu(input_x)
>>> print(output)
[[-1.1113307 4.202804 -1.7575096]
[ 2.101402 -1.7462534 9.456309 ]]
```

mindspore.ops.sigmoid

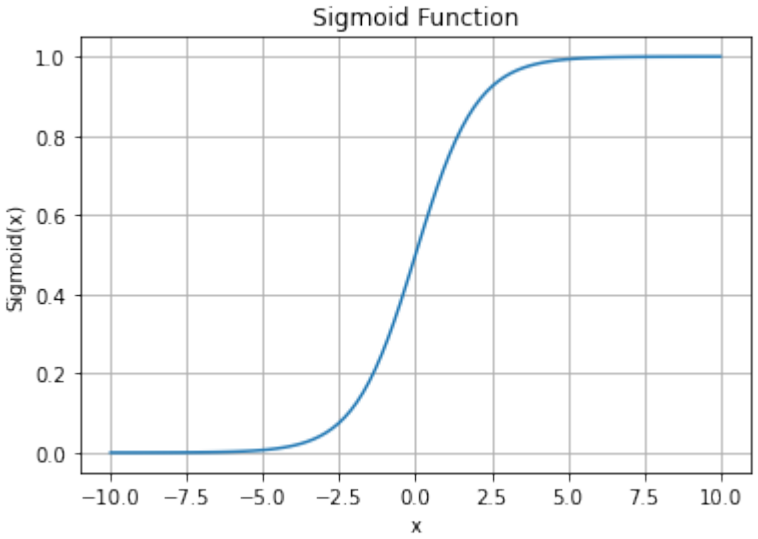
`mindspore.ops.`**`sigmoid`**(*input*)

Computes Sigmoid of input element-wise. The Sigmoid function is defined as:

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)}$$

where x_i is an element of x.

Sigmoid Function Graph:



Parameters

input (Tensor) – *input* is x in the preceding formula. Tensor of any dimension, the data type is float16, float32, float64, complex64 or complex128.

Returns

Tensor, with the same type and shape as the input.

Raises

- **TypeError** – If dtype of *input* is not float16, float32, float64, complex64 or complex128.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = ops.sigmoid(input)
>>> print(output)
[0.7310586  0.880797   0.95257413 0.98201376 0.9933072 ]
```

mindspore.ops.silu

`mindspore.ops.`**`silu`**(*input*)

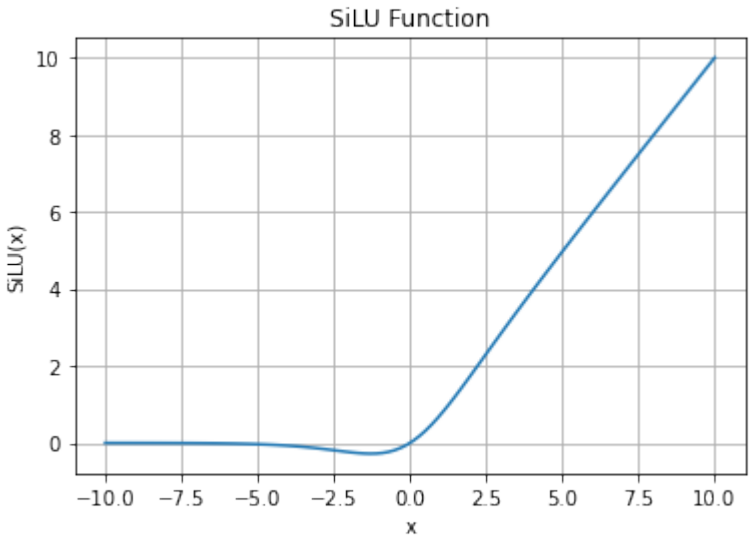
Computes Sigmoid Linear Unit of input element-wise, also known as Swish function. The SiLU function is defined as:

$$\text{SiLU}(x) = x * \sigma(x),$$

where x is an element of the input, $\sigma(x)$ is Sigmoid function.

$$\text{sigma}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

SiLU Function Graph:



Parameters

input (Tensor) – *input* is x in the preceding formula. Input with the data type float16 or float32.

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> output = ops.silu(input)
>>> print(output)
[-0.269  1.762  -0.1423  1.762  -0.269]
```

mindspore.ops.softmax

mindspore.ops.**softmax**(*input*, *axis=- 1*, ***, *dtype=None*)

Applies the Softmax operation to the input tensor on the specified axis. Suppose a slice in the given axis *axis*, then for each element $input_i$, the Softmax function is shown as follows:

$$\text{output}(input_i) = \frac{\exp(input_i)}{\sum_{j=0}^{N-1}\exp(input_j)},$$

where N is the length of the tensor.

Parameters

- **input** (Tensor) – Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions, with float16 or float32 data type.
- **axis** (*int, optional*) – The axis to perform the Softmax operation. Default: `-1` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) – When set, *input* will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. Default: `None` .

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If *axis* is not an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = ops.softmax(input)
>>> print(output)
[0.01165623 0.03168492 0.08612854 0.23412167 0.6364086 ]
```

mindspore.ops.softmin

mindspore.ops.**softmin**(*x*, *axis=- 1*, ***, *dtype=None*)

Applies the Softmin operation to the input tensor on the specified axis. Suppose a slice in the given axis x, then for each element x_i, the Softmin function is shown as follows:

$$\text{output}(x_i) = \frac{\exp(-x_i)}{\sum_{j=0}^{N-1}\exp(-x_j)},$$

where N is the length of the tensor.

Parameters

- **axis** (*Union[int, tuple[int]], optional*) – The axis to perform the Softmin operation. Default: `-1` .
- **x** (`Tensor`) – Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions, with float16 or float32 data type.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – When set, x will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. Default: `None` .

Returns

Tensor, with the same type and shape as x.

Raises

- **TypeError** – If *axis* is not an int or a tuple.
- **TypeError** – If dtype of x is neither float16 nor float32.
- **ValueError** – If *axis* is a tuple whose length is less than 1.
- **ValueError** – If *axis* is a tuple whose elements are not all in range [-len(logits.shape), len(logits.shape)).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> output = ops.softmin(x)
>>> print(output)
[0.2341  0.636  0.0862  0.01165  0.03168 ]
```

mindspore.ops.softshrink

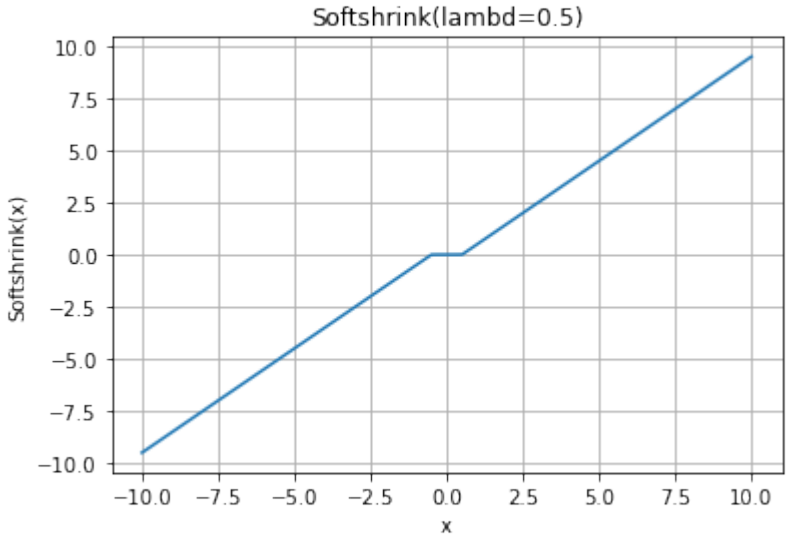
mindspore.ops.**softshrink**(*input*, *lambd=0.5*)

Soft Shrink activation function. Calculates the output according to the input elements.

The formula is defined as follows:

$$\text{SoftShrink}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

SoftShrink Activation Function Graph:



Parameters

- **input** (Tensor) – The input of Soft Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32.
- **lambd** (*number, optional*) – The threshold λ defined by the Soft Shrink formula. It should be greater than or equal to 0, default: 0.5 .

Returns

Tensor, has the same data type and shape as the input *input*.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import numpy as np
>>> x = Tensor(np.array([[ 0.5297,  0.7871,  1.1754], [ 0.7836,  0.6218, -1.1542]]),␣
↪mindspore.float32)
>>> output = ops.softshrink(x)
>>> print(output)
[[ 0.02979  0.287    0.676  ]
[ 0.2837   0.1216  -0.6543 ]]
```

mindspore.ops.softsign

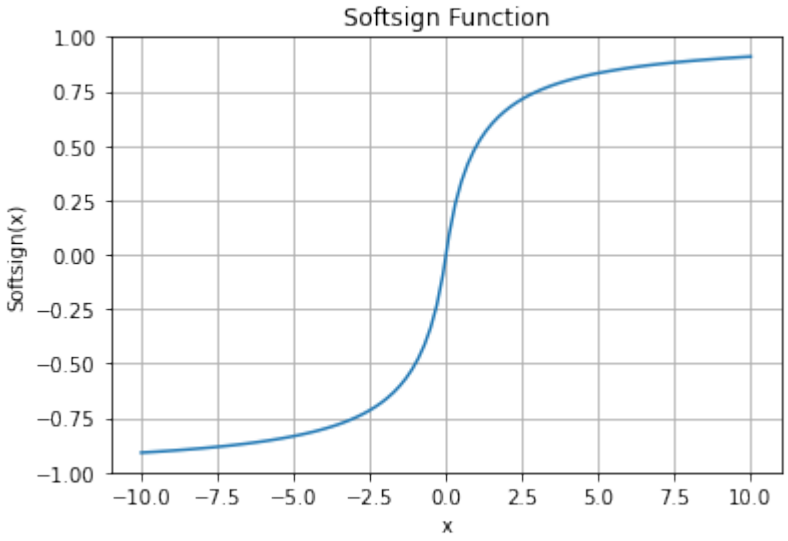
mindspore.ops.**softsign**(*x*)

SoftSign activation function.

The function is shown as follows:

$$\text{SoftSign}(x) = \frac{x}{1 + |x|}$$

Softsign Activation Function Graph:



Parameters

x (Tensor) – Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions, with float16 or float32 data type.

Returns

Tensor, with the same type and shape as the *x*.

Raises

- **TypeError** –If *x* is not a Tensor.
- **TypeError** –If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0, -1, 2, 30, -30]), mindspore.float32)
>>> output = ops.softsign(x)
>>> print(output)
[ 0.        -0.5         0.6666667  0.9677419 -0.9677419]
```

mindspore.ops.swiglu

mindspore.ops.**swiglu**(*input*, *dim=- 1*)

Computes SwiGLU (Swish-Gated Linear Unit activation function) of input tensor. SwiGLU is a variant of the *mindspore.ops.GLU* activation function, it is defined as:

> **Warning:** This is an experimental API that is subject to change or deletion.

$$SwiGLU(a, b) = Swish(a) \otimes b$$

where a is the first half of the *input* matrices and b is the second half, Swish(a)=a σ (a), σ is the *mindspore.ops.sigmoid()* activation function and $\otimes$ is the Hadamard product.

Parameters

- **input** (Tensor) –Tensor to be split. It has shape $(*_1, N, *_2)$ where * means, any number of additional dimensions. N must be divisible by 2.
- **dim** (*int, optional*) –the axis to split the input. It must be int. Default: `-1` , the last axis of *input*.

Returns

Tensor, the same dtype as the *input*, with the shape $(*_1, M, *_2)$ where $M = N/2$.

Raises

- **TypeError** –If dtype of *input* is not float16, float32 or bfloat16.
- **TypeError** –If *input* is not a Tensor.
- **RuntimeError** –If the dimension specified by *dim* is not divisible by 2.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input = Tensor([[-0.12, 0.123, 31.122], [2.1223, 4.1212121217, 0.3123]], dtype=mindspore.
↪float32)
>>> output = ops.swiglu(input, 0)
>>> print(output)
[[-0.11970687 0.2690224 9.7194 ]]
```

mindspore.ops.tanh

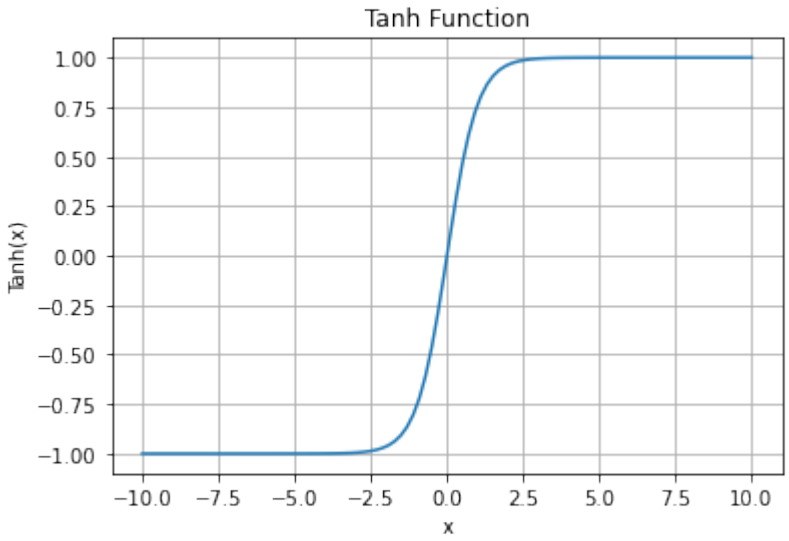
`mindspore.ops.`**`tanh`**(*input*)

Computes hyperbolic tangent of input element-wise. The Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Tanh Activation Function Graph:



Parameters

`input` (`Tensor`) – Input of Tanh.

Returns

Tensor, with the same type and shape as the *input*.

Raises

`TypeError` – If *input* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = ops.tanh(input)
>>> print(output)
[0.7615941 0.9640276 0.9950547 0.9993293 0.9999092]
```

mindspore.ops.threshold

mindspore.ops.**threshold**(*input*, *thr*, *value*)

Returns each element of *input* after thresholding by *thr* as a Tensor.

The formula is defined as follows:

$$y = \begin{cases} input, & \text{if } input > \text{thr} \\ \text{value}, & \text{otherwise} \end{cases}$$

Parameters

- **input** (Tensor) –The input of threshold with data type of float16 or float32.
- **thr** (*Union[int, float]*) –The value of the threshold.
- **value** (*Union[int, float]*) –The value to replace with when element is less than threshold.

Returns

Tensor, the same shape and data type as the input.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *thr* is not a float or an int.
- **TypeError** –If *value* is not a float or an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> inputs = mindspore.Tensor([0.0, 2, 3], mindspore.float32)
>>> outputs = ops.threshold(inputs, 1, 100)
>>> print(outputs)
[100.   2.   3.]
```

2.3.4 Distance Functions

API Name	Description	Supported Platforms
mindspore.ops.cdist	Computes p-norm distance between each pair of row vectors of two input Tensors.	Ascend GPU CPU
mindspore.ops.dist	Computes batched the *p*-norm distance between each pair of the two collections of row vectors.	Ascend GPU CPU
mindspore.ops.pdist	Calculates the distance between every pair of row vectors in the input using the p-norm.	GPU CPU

mindspore.ops.cdist

mindspore.ops.**cdist**(*x1*, *x2*, *p=2.0*)

Computes p-norm distance between each pair of row vectors of two input Tensors.

Note:

- On Ascend, the supported dtypes are float16 and float32.
- On CPU, the supported dtypes are float16 and float32.
- On GPU, the supported dtypes are float32 and float64.

Parameters

- **x1** (Tensor) −Input tensor of shape (B, P, M). Letter B represents 0 or positive int number. When B is equal to 0, it means this dimension can be ignored, i.e. shape of the tensor is (P, M).
- **x2** (Tensor) −Input tensor of shape (B, R, M), has the same dtype as *x1*.
- **p** (*float, optional*) −P value for the p-norm distance to calculate between each vector pair, P >= 0. Default: 2.0.

Returns

Tensor, p-norm distance, has the same dtype as *x1*, its shape is (B, P, R).

Raises

- **TypeError** −If *x1* or *x2* is not Tensor.
- **TypeError** −If dtype of *x1* or *x2* is not listed in the "Note" above.
- **TypeError** −If *p* is not float32.
- **ValueError** −If *p* is negative.
- **ValueError** −If dimension of *x1* is not the same as *x2*.
- **ValueError** −If dimension of *x1* or *x2* is neither 2 nor 3.
- **ValueError** −If the batch dim of *x1* and *x2* can not broadcast.
- **ValueError** −If the number of columns of *x1* is not the same as that of *x2*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[1.0, 1.0], [2.0, 2.0]]]).astype(np.float32))
>>> y = Tensor(np.array([[[3.0, 3.0], [3.0, 3.0]]]).astype(np.float32))
>>> output = ops.cdist(x, y, 2.0)
>>> print(output)
[[[2.8284273 2.8284273]
  [1.4142137 1.4142137]]]
```

mindspore.ops.dist

mindspore.ops.**dist**(*input*, *other*, *p=2*)

Computes batched the *p*-norm distance between each pair of the two collections of row vectors.

Note: Since only normalization for integer *p*-normal form is supported in MindSpore, a type error will be raised if *p* is not an integer.

Parameters

- **input** (Tensor) –The first input tensor. The dtype must be float16 or float32.
- **other** (Tensor) –The second input tensor. The dtype must be float16 or float32.
- **p** (*int, optional*) –The order of norm. *p* is greater than or equal to 0. Default: 2 .

Returns

Tensor, has the same dtype as *input*, which shape is (1).

Raises

- **TypeError** –If *input* or *other* is not a Tensor.
- **TypeError** –If dtype of *input* or *other* is neither float16 nor float32.
- **TypeError** –If *p* is not a non-negative integer.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor, ops
>>> input_x = Tensor([[[1.0, 1.0], [2.0, 2.0]]])
>>> input_y = Tensor([[[3.0, 3.0], [3.0, 3.0]]])
>>> out = ops.dist(input_x, input_y)
>>> print(out.asnumpy())
3.1622777
```

mindspore.ops.pdist

mindspore.ops.**pdist**(*input*, *p=2.0*)

Calculates the distance between every pair of row vectors in the input using the p-norm. If the input *input* is a 2D Tensor with shape (N, M), the *output* must be a 1D Tensor with shape $(N * (N-1)/2,)$.

$$y[n] = \sqrt[p]{| x_i - x_j |^p}$$

where x_i, x_j are two different row vectors in the input.

Parameters

- **input** (Tensor) –Input tensor. dtype: float16, float32 or float64.
- **p** (*float*) –The order of norm distance, p [0,). Default: 2.0 .

Returns

Tensor, has the same dtype as *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **TypeError** –If *p* is not a float.
- **ValueError** –If *p* is a negative float.
- **ValueError** –If dimension of *input* is less than 2.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]]).astype(np.float32))
>>> y = ops.pdist(x, p=2.0)
>>> print(y)
[1.4142135 2.828427 1.4142135]
```

2.3.5 Sampling Functions

API Name	Description	Supported Platforms
mindspore.ops.choice_with_mask	Generates a random sample as index tensor with a mask tensor from a given tensor.	Ascend GPU CPU
mindspore.ops.random_categorical	Generates random samples from a given categorical distribution tensor.	Ascend GPU CPU
mindspore.ops.log_uniform_candidate_sampler	Generates random labels with a log-uniform distribution for sampled_candidates.	Ascend CPU
mindspore.ops.uniform_candidate_sampler	Uniform candidate sampler.	Ascend GPU CPU

mindspore.ops.choice_with_mask

mindspore.ops.**choice_with_mask**(*input_x*, *count=256*, *seed=None*)

Generates a random sample as index tensor with a mask tensor from a given tensor.

The *input_x* must be a tensor whose dimension is not less than 1. If its dimension is greater than or equal to 2, the first dimension specifies the number of samples. The returned index tensor denotes the index of the nonzero sample, the mask tensor denotes which elements in the index tensor are valid.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **input_x** (*Tensor[bool]*) – The input tensor. The input tensor rank must be greater than or equal to 1 and less than or equal to 5.
- **count** (*int, optional*) – Number of items expected to get and the number must be greater than 0. Default: 256 .
- **seed** (*int, optional*) – Seed is used as entropy source for Random number engines generating pseudo-random numbers. Default: None .

Returns

Two tensors, the first one is the index tensor and the other one is the mask tensor.

- **index** (Tensor) - The output shape is 2-D.
- **mask** (Tensor) - The output shape is 1-D.

Raises

- **TypeError** – If *count* is not an int.
- **TypeError** – If *seed* is not an int.
- **TypeError** – If *input_x* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones(shape=[240000, 4]).astype(np.bool_))
>>> output_y, output_mask = ops.choice_with_mask(input_x)
>>> result = output_y.shape
>>> print(result)
(256, 2)
>>> result = output_mask.shape
>>> print(result)
(256,)
```

mindspore.ops.random_categorical

mindspore.ops.**random_categorical**(*logits*, *num_sample*, *seed=0*, *dtype=mstype.int64*)

Generates random samples from a given categorical distribution tensor.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **logits** (Tensor) –The input tensor. 2-D Tensor with shape $(batch\_size, num\_classes)$.
- **num_sample** (*int*) –Number of sample to be drawn. Only constant values is allowed.
- **seed** (*int*) –Random seed. Only constant values is allowed. Default: `0` .
- **dtype** (mindspore.dtype) –The type of output. Its value must be one of mindspore.int16, mindspore.int32 and mindspore.int64. Default: `mstype.int64` .

Returns

Tensor, The output Tensor with shape $(batch\_size, num\_samples)$.

Raises

- **TypeError** –If *dtype* is not one of the following: mindspore.int16, mindspore.int32, mindspore.int64.
- **TypeError** –If *logits* is not a Tensor.
- **TypeError** –If neither *num_sample* nor *seed* is an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> import mindspore.common.dtype as mstype
>>> import numpy as np
>>> logits = Tensor(np.random.random((10, 5)).astype(np.float32), mstype.float32)
>>> net = ops.random_categorical(logits, 8)
>>> result = net.shape
>>> print(result)
(10, 8)
```

mindspore.ops.log_uniform_candidate_sampler

mindspore.ops.**log_uniform_candidate_sampler**(*true_classes*, *num_true=1*, *num_sampled=5*, *unique=True*, *range_max=5*, *seed=0*)

Generates random labels with a log-uniform distribution for sampled_candidates.

Randomly samples a tensor of sampled classes from the range of integers [0, range_max).

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **true_classes** (Tensor) –The target classes. With data type of int64 and shape $(batch\_size, num\_true)$.
- **num_true** (*int, optional*) –The number of target classes per training example. Default: `1` .
- **num_sampled** (*int, optional*) –The number of classes to randomly sample. Default: `5` .
- **unique** (*bool, optional*) –Determines whether sample with rejection. If *unique* is `True` , all sampled classes in a batch are unique. Default: `True` .
- **range_max** (*int, optional*) –The number of possible classes. When *unique* is `True` , *range_max* must be greater than or equal to *num_sampled*. Default: `5` .
- **seed** (*int, optional*) –Random seed, must be non-negative. Default: `0` .

Returns

Tuple of 3 Tensors.

- **sampled_candidates** (Tensor) - A Tensor with shape $(num\_sampled,)$ and the same type as *true_classes*.
- **true_expected_count** (Tensor) - A Tensor with the same shape as *true_classes and* type float32.
- **sampled_expected_count** (Tensor) - A Tensor with the same shape as *sampled_candidates* and type float32.

Raises

- **TypeError** –If neither *num_true* nor *num_sampled* is an int.
- **TypeError** –If *unique* is not a bool.
- **TypeError** –If neither *range_max* nor *seed* is an int.
- **TypeError** –If *true_classes* is not a Tensor.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> output1, output2, output3 = ops.log_uniform_candidate_sampler(
... Tensor(np.array([[1, 7], [0, 4], [3, 3]])), 2, 5, True, 5)
>>> print(output1, output2, output3)
[3 2 0 4 1]
[[0.92312991 0.49336370]
 [0.99248987 0.65806371]
 [0.73553443 0.73553443]]
[0.73553443 0.82625800 0.99248987 0.65806371 0.92312991]
```

mindspore.ops.uniform_candidate_sampler

mindspore.ops.**uniform_candidate_sampler**(*true_classes*, *num_true*, *num_sampled*, *unique*, *range_max*, *seed=0*, *remove_accidental_hits=False*)

Uniform candidate sampler.

This function samples a set of classes(sampled_candidates) from [0, range_max-1] based on uniform distribution. If unique=True, candidates are drawn without replacement, else unique=False with replacement.

Warning:

- The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.
- The Ascend backend does not support dynamic shape scenarios currently.

Parameters

- **true_classes** (Tensor) −A Tensor. The target classes with a Tensor shape of $(batch\_size, num\_true)$. The value range of the elements must be $[0, range\_max)$.
- **num_true** (*int*) −The number of target classes in each training example.
- **num_sampled** (*int*) −The number of classes to randomly sample. The sampled_candidates will have a shape of num_sampled. If unique=True, num_sampled must be less than or equal to range_max.
- **unique** (*bool*) −Whether all sampled classes in a batch are unique.
- **range_max** (*int*) −The number of possible classes, must be positive.
- **seed** (*int*) −Used for random number generation, must be non-negative. If seed has a value of 0, the seed will be replaced with a randomly generated value. Default: `0` .
- **remove_accidental_hits** (*bool*) −Whether accidental hit is removed. Accidental hit is when one of the true classes matches one of the sample classes. Set `True` to remove which accidentally sampling the true class as sample class. Default: `False` .

Returns

- **sampled_candidates** (Tensor) - The sampled_candidates is independent of the true classes. shape: $(num\_sampled,)$.
- **true_expected_count** (Tensor) - The expected counts under the sampling distribution of each of true_classes. shape: $(batch\_size, num\_true)$.
- **sampled_expected_count** (Tensor) - The expected counts under the sampling distribution of each of sampled_candidates. shape: $(num\_sampled,)$.

Raises

- **TypeError** −If neither *num_true* nor *num_sampled* is an int.
- **TypeError** −If neither *unique* nor *remove_accidental_hits* is a bool.
- **TypeError** −If neither *range_max* nor *seed* is an int.
- **TypeError** −If *true_classes* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> data = Tensor(np.array([[1], [3], [4], [6], [3]], dtype=np.int64))
>>> output1, output2, output3 = ops.uniform_candidate_sampler(data, 1, 3, False, 4, 1)
>>> print(output1.shape)
(3,)
>>> print(output2.shape)
(5, 1)
>>> print(output3.shape)
(3,)
```

2.3.6 Image Functions

API Name	Description	Supported Platforms
mindspore.ops.affine_grid	Returns a 2D or 3D flow field (sampling grid) based on *theta*, a batch of affine matrices.	Ascend GPU CPU
mindspore.ops.bounding_box_decode	Decode the bounding box locations, calculate the offset, and convert the offset into a Bbox, which is used to mark the target in the subsequent images, etc.	Ascend GPU CPU
mindspore.ops.bounding_box_encode	Encode the bounding box locations, calculate the offset between the predicted bounding boxes and the real bounding boxes, and the offset will be used as a variable for the loss.	Ascend GPU CPU
mindspore.ops.col2im	Combines an array of sliding local blocks into a large containing tensor.	Ascend GPU CPU
mindspore.ops.check_valid	Checks whether the bounding box is in the image.	Ascend GPU CPU
mindspore.ops.crop_and_resize	Extracts crops from the input image Tensor and resizes them.	Ascend GPU CPU
mindspore.ops.grid_sample	Given an *input* and a flow-field *grid*, computes the *output* using *input* values and pixel locations from *grid*.	Ascend GPU CPU
mindspore.ops.interpolate	Samples the input Tensor to the given size or scale_factor by using one of the interpolate algorithms.	Ascend GPU CPU
mindspore.ops.iou	Calculates intersection over union for boxes.	Ascend GPU CPU
mindspore.ops.pad	Pads the input tensor according to the padding.	Ascend GPU CPU
mindspore.ops.padding	Extends the last dimension of the input tensor from 1 to pad_dim_size, by filling with 0.	Ascend GPU CPU
mindspore.ops.pixel_shuffle	Applies the PixelShuffle operation over input *input* which implements sub-pixel convolutions with stride $1/r$.	Ascend GPU CPU
mindspore.ops.pixel_unshuffle	Applies the PixelUnshuffle operation over input *input* which is the inverse of PixelShuffle.	Ascend GPU CPU
mindspore.ops.rotary_position_embedding	Implements the Rotary Position Embedding algorithm.	Ascend
mindspore.ops.rotated_iou	Calculate the overlap area between rotated rectangles.	Ascend

continues on next page

Table 16 – continued from previous page

mindspore.ops.upsample	Alias for *mindspore.ops.interpolate()*.	Ascend GPU CPU

mindspore.ops.affine_grid

mindspore.ops.**affine_grid**(*theta*, *size*, *align_corners=False*)

Returns a 2D or 3D flow field (sampling grid) based on *theta*, a batch of affine matrices.

Parameters

- **theta** (Tensor) –The input tensor of flow field whose dtype is float16, float32. Input batch of affine matrices with shape $(N, 2, 3)$ for 2D grid or $(N, 3, 4)$ for 3D grid.
- **size** (*tuple[int]*) –The target output image size. The value of target output with format (N, C, H, W) for 2D grid or (N, C, D, H, W) for 3D grid.
- **align_corners** (*bool, optional*) –Geometrically, each pixel of input is viewed as a squqre instead of dot. If True , consider extremum -1 and 1 referring to the centers of the pixels rather than pixel corners. The default value is False , extremum -1 and 1 refer to the corners of the pixels, so that sampling is irrelevant to resolution of the image. Default: False .

Returns

Tensor, a tensor whose data type is same as 'theta', and the shape is $(N, H, W, 2)$ for 2D grid or $(N, D, H, W, 3)$ for 3D grid.

Raises

- **TypeError** –If *theta* is not a Tensor or *size* is not a tuple.
- **ValueError** –If the shape of *theta* is not $(N, 2, 3)$ or $(N, 3, 4)$.
- **ValueError** –If the size of *size* is not 4 or 5.
- **ValueError** –If the shape of *theta* is $(N, 2, 3)$, the size of *size* is not 4; If the shape of *theta* is $(N, 3, 4)$, the size of *size* is not 5.
- **ValueError** –If the size[0] is not equal to the shape[0] of theta.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> theta = Tensor([[[0.8, 0.5, 0],[-0.5, 0.8, 0]]], mindspore.float32)
>>> out_size = (1, 3, 2, 3)
>>> output = ops.affine_grid(theta, out_size, False)
>>> print(output)
[[[[-0.78333336 -0.06666666]
[-0.25       -0.4       ]
[ 0.28333336 -0.73333335]]
[[-0.28333336  0.73333335]
[ 0.25        0.4       ]
[ 0.78333336  0.06666666]]]]
```

mindspore.ops.bounding_box_decode

mindspore.ops.**bounding_box_decode**(*anchor_box*, *deltas*, *max_shape*, *means=(0.0, 0.0, 0.0, 0.0)*, *stds=(1.0, 1.0, 1.0, 1.0)*, *wh_ratio_clip=0.016*)

Decode the bounding box locations, calculate the offset, and convert the offset into a Bbox, which is used to mark the target in the subsequent images, etc.

Parameters

- **anchor_box** (Tensor) –Anchor boxes. The shape of *anchor_box* must be $(n, 4)$.
- **deltas** (Tensor) –Delta of boxes. Which has the same shape with *anchor_box*.
- **max_shape** (*tuple*) –The max size limit for decoding box calculation.
- **means** (*tuple, optional*) –The means of *deltas* calculation. Default: `(0.0, 0.0, 0.0, 0.0)`.
- **stds** (*tuple, optional*) –The standard deviations of *deltas* calculation. Default: `(1.0, 1.0, 1.0, 1.0)`.
- **wh_ratio_clip** (*float, optional*) –The limit of width and height ratio for decoding box calculation. Default: `0.016`.

Returns

Tensor, decoded boxes. It has the same data type and shape as *anchor_box*.

Raises

- **TypeError** –If *means*, *stds* or *max_shape* is not a tuple.
- **TypeError** –If *wh_ratio_clip* is not a float.
- **TypeError** –If *anchor_box* or *deltas* is not a Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> anchor_box = Tensor([[4, 1, 2, 1], [2, 2, 2, 3]], mindspore.float32)
>>> deltas = Tensor([[3, 1, 2, 2], [1, 2, 1, 4]], mindspore.float32)
>>> output = ops.bounding_box_decode(anchor_box, deltas, max_shape=(768, 1280), means=(0.0,␣
↪0.0, 0.0, 0.0),
...                                  stds=(1.0, 1.0, 1.0, 1.0), wh_ratio_clip=0.016)
>>> print(output)
[[ 4.1953125  0.         0.         5.1953125]
 [ 2.140625   0.         3.859375  60.59375  ]]
```

mindspore.ops.bounding_box_encode

mindspore.ops.**bounding_box_encode**(*anchor_box*, *groundtruth_box*, *means=(0.0, 0.0, 0.0, 0.0)*, *stds=(1.0, 1.0, 1.0, 1.0)*)

Encode the bounding box locations, calculate the offset between the predicted bounding boxes and the real bounding boxes, and the offset will be used as a variable for the loss.

Parameters

- **anchor_box** (Tensor) – Anchor boxes. The shape of *anchor_box* must be $(n, 4)$.
- **groundtruth_box** (Tensor) – Ground truth boxes. Which has the same shape with *anchor_box*.
- **means** (*tuple, optional*) – Means for encoding bounding boxes calculation. Default: `(0.0, 0.0, 0.0, 0.0)`.
- **stds** (*tuple, optional*) – The standard deviations of deltas calculation. Default: `(1.0, 1.0, 1.0, 1.0)`.

Returns

Tensor, encoded bounding boxes. It has the same data type and shape as input *anchor_box*.

Raises

- **TypeError** – If *means* or *stds* is not a tuple.
- **TypeError** – If *anchor_box* or *groundtruth_box* is not a Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> anchor_box = Tensor([[2, 2, 2, 3], [2, 2, 2, 3]], mindspore.float32)
>>> groundtruth_box = Tensor([[1, 2, 1, 4], [1, 2, 1, 4]], mindspore.float32)
>>> output = ops.bounding_box_encode(anchor_box, groundtruth_box, means=(0.0, 0.0, 0.0, 0.0),
...                                  stds=(1.0, 1.0, 1.0, 1.0))
>>> print(output)
[[ -1.   0.25  0.   0.40551758]
 [ -1.   0.25  0.   0.40551758]]
```

mindspore.ops.col2im

mindspore.ops.**col2im**(*input_x*, *output_size*, *kernel_size*, *dilation*, *padding_value*, *stride*)

Combines an array of sliding local blocks into a large containing tensor.

Parameters

- **input_x** (Tensor) – 4D tensor with data type float16 or float32.
- **output_size** (Tensor) – 1D tensor with 2 elements of data type int.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) – The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width. Must be specified.
- **dilation** (*Union[int, tuple[int], list[int]]*) – The size of the dilation, should be two int for height and width. If type is int, it means that height equal with width.

- **padding_value** (*Union[int, tuple[int], list[int]]*) –The size of the padding, should be two int for height and width. If type is int, it means that height equal with width.
- **stride** (*Union[int, tuple[int], list[int]]*) –The size of the stride, should be two int for height and width. If type is int, it means that height equal with width.

Returns

A 4D Tensor, with same type as *input_x*.

Raises

- **TypeError** –If `kernel_size`, *dilation*, *padding_value*, *stride* data type is not in Union[int, tuple[int], list[int]].
- **ValueError** –If `kernel_size`, *dilation*, *padding_value*, *stride* value is not greater than zero or elements number more than 2.
- **ValueError** –If `padding_value` value is less than zero or elements number more than 2.
- **ValueError** –If input_x.shape[2] != kernel_size[0] * kernel_size[1].
- **ValueError** –If input_x.shape[3] does not match the calculated number of sliding blocks.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(input_data=np.random.rand(16, 16, 4, 25), dtype=mstype.float32)
>>> output_size = Tensor(input_data=[8, 8], dtype=mstype.int32)
>>> output = ops.col2im(x, output_size, [2, 2], [2, 2], [2, 2], [2, 2])
>>> print(output.shape)
(16, 16, 8, 8)
```

mindspore.ops.check_valid

mindspore.ops.**check_valid**(*bboxes*, *img_metas*)

Checks whether the bounding box is in the image.

bboxes contain several sets of bounding boxes, each represented by two abscissa points $(x0, x1)$ and two ordinate points $(y0, y1)$. *img_metas* provides information about the original image, including three parameters $(height, width, ratio)$, which specify the valid boundary of the image.

when the following conditions are met:

$x0 >= 0$

$y0 >= 0$

$x1 <= width * ratio - 1$

$y1 <= height * ratio - 1$

the bounding box is considered to be within the image.

Warning: The bounding box specified by *bboxes* and the image information specified by *img_metas* need to be valid, i.e.: $x0 <= x1$, $y0 <= y1$, and $(height, width, ratio)$ are all positive.

Parameters

- **bboxes** (Tensor) −Bounding boxes tensor with shape $(N, 4)$. N indicates the number of bounding boxes, the value *4* indicates four coordinate points $(x0, y0, x1, y1)$. Data type must be float16 or float32.
- **img_metas** (Tensor) −Raw image size information with the format of $(height, width, ratio)$, specifying the valid boundary $(height * ratio - 1, width * ratio - 1)$. Data type must be float16 or float32.

Returns

Tensor, with shape of $(N,)$ and dtype of bool, specifying whether the bounding boxes is in the image. *True* indicates valid, while *False* indicates invalid.

Raises

- **TypeError** −If *bboxes* or *img_metas* is not a Tensor.
- **TypeError** −If dtype of *bboxes* or *img_metas* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bboxes = Tensor(np.linspace(0, 6, 12).reshape(3, 4), mindspore.float32)
>>> img_metas = Tensor(np.array([2, 1, 3]), mindspore.float32)
>>> output = ops.check_valid(bboxes, img_metas)
>>> print(output)
[ True False False]
```

mindspore.ops.crop_and_resize

mindspore.ops.**crop_and_resize**(*image*, *boxes*, *box_indices*, *crop_size*, *method='bilinear'*, *extrapolation_value=0.0*)

Extracts crops from the input image Tensor and resizes them.

Note: In case that the output shape depends on crop_size, the crop_size must be constant. For now, the backward of the operator only supports bilinear method, for other methods, will return 0.

Parameters

- **image** (Tensor) −A 4-D Tensor representing a batch of images. It has shape $(batch, image\_height, image\_width, depth)$.
- **boxes** (Tensor) −A 2-D Tensor with shape $(num\_boxes, 4)$ representing the normalized coordinates of the boxes to be cropped. The coordinates are specified in the form $[y1, x1, y2, x2]$, where $(y1, x1)$ is the first corner and $(y2, x2)$ is the second corner of the box. If $y1 > y2$, the sampled crop is inverted upside down, the width dimensionis treated

similarly when $x1 > x2$. If normalized coordinates are not in range $[0, 1]$, extrapolated input image values are used instead. Supported data type: float32.

- **box_indices** (Tensor) −A 1-D Tensor of shape (*num_boxes*) representing the batch index for each box. Supported type: int32.
- **crop_size** (*Tuple[int]*) −A tuple of two elements: (crop_height, crop_width), representing the output size of the cropped and resized images. Only positive values are supported. Supported type: int32.
- **method** (*str, optional*) −An optional string that specifies the sampling method for resizing. It can be `"bilinear"`, `"nearest"` or `"bilinear_v2"`. Default: `"bilinear"`.
  - `"nearest"`: Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.
  - `"bilinear"`: Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels, computed using bilinear interpolation. This method produces smoother results compared to nearest neighbor interpolation.
  - `"bilinear_v2"`: The optimized variant of `"bilinear"`, it may achieve better result(higher precision and speed) in some cases.
- **extrapolation_value** (*float, optional*) −An optional float value used extrapolation, if applicable. Default: `0.0`.

Returns

A 4-D tensor of shape ($num\_boxes, crop\_height, crop\_width, depth$) with type(float32).

Raises

- **TypeError** −If *image* or *boxes* or *box_indices* is not a Tensor.
- **TypeError** −If *crop_size* is not a Tuple with two int32 elements.
- **TypeError** −If dtype of *boxes* is not float or that of *box_indices* is not int32.
- **TypeError** −If *method* is not a str.
- **TypeError** −If *extrapolation_value* is not a float.
- **ValueError** −If the shape rank of *image* is not 4.
- **ValueError** −If the shape rank of *boxes* is not 2.
- **ValueError** −If the second dim of *boxes* is not 4.
- **ValueError** −If the shape rank of *box_indices* is not 1.
- **ValueError** −If the first dim of *box_indices* is not equal to that of *boxes*.
- **ValueError** −If existing element in *box_indices* is out of range *[0, batch)*.
- **ValueError** −If the data of *crop_size* is not positive.
- **ValueError** −If *method* is not one of 'bilinear', 'nearest', 'bilinear_v2'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import ops, Tensor
>>> BATCH_SIZE = 1
>>> NUM_BOXES = 5
>>> IMAGE_HEIGHT = 256
>>> IMAGE_WIDTH = 256
>>> CHANNELS = 3
>>> image = np.random.normal(size=[BATCH_SIZE, IMAGE_HEIGHT, IMAGE_WIDTH, CHANNELS]).
↪astype(np.float32)
>>> boxes = np.random.uniform(size=[NUM_BOXES, 4]).astype(np.float32)
>>> box_indices = np.random.uniform(size=[NUM_BOXES], low=0, high=BATCH_SIZE).astype(np.
↪int32)
>>> crop_size = (24, 24)
>>> output = ops.crop_and_resize(Tensor(image), Tensor(boxes), Tensor(box_indices), crop_
↪size)
>>> print(output.shape)
(5, 24, 24, 3)
```

mindspore.ops.grid_sample

mindspore.ops.**grid_sample**(*input*, *grid*, *mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*)

Given an *input* and a flow-field *grid*, computes the *output* using *input* values and pixel locations from *grid*. Only spatial (4-D) and volumetric (5-D) *input* is supported.

In the spatial (4-D) case, for *input* with shape (N, C, H_{in}, W_{in}) and *grid* with shape $(N, H_{out}, W_{out}, 2)$, the *output* will have shape (N, C, H_{out}, W_{out}).

For each output location *output[n, :, h, w]*, the size-2 vector *grid[n, h, w]* specifies *input* pixel locations x and y, which are used to interpolate the output value *output[n, :, h, w]*. In the case of 5D inputs, *grid[n, d, h, w]*, specifies the x, y, z pixel locations for interpolating *output[n, :, d, h, w]*. And *mode* argument specifies "nearest" or "bilinear" ("bicubic" is not supported yet) interpolation method to sample the input pixels.

grid specifies the sampling pixel locations normalized by the *input* spatial dimensions. Therefore, it should have most values in the range of $[-1, 1]$.

If *grid* has values outside the range of $[-1, 1]$, the corresponding outputs are handled as defined by *padding_mode*. If *padding_mode* is set to be "zeros", use 0 for out-of-bound grid locations. If *padding_mode* is set to be "border", use border values for out-of-bound grid locations. If *padding_mode* is set to be "reflection", use values at locations reflected by the border for out-of-bound grid locations. For location far away from the border, it will keep being reflected until becoming in bound.

Parameters

- **input** (Tensor) – input with shape of (N, C, H_{in}, W_{in}) (4-D case) or $(N, C, D_{in}, H_{in}, W_{in})$ (5-D case) and dtype of float32 or float64.
- **grid** (Tensor) – flow-field with shape of $(N, H_{out}, W_{out}, 2)$ (4-D case) or $(N, D_{out}, H_{out}, W_{out}, 3)$ (5-D case) and same dtype as *input*.
- **mode** (*str, optional*) – An optional string specifying the interpolation method. The optional values are `'bilinear'`, `'nearest'`. Default: `'bilinear'` . Note: *bicubic* is not supported yet. When *mode="bilinear"* and the input is 5-D, the interpolation mode used internally will actually be trilinear. However, when the input is 4-D, the interpolation mode will legistimately be bilinear. Default: `'bilinear'` .
  - `'nearest'`: Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.

  - `'bilinear'`: Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels, computed using bilinear interpolation. This method produces smoother results compared to nearest neighbor interpolation.
  - `'trilinear'`: Trilinear interpolation. This is an extension of bilinear interpolation to 3D data. It performs bilinear interpolation in the two spatial dimensions and linear interpolation along the third dimension. It is commonly used for volume or 3D image interpolation.

- **padding_mode** (*str, optional*) – An optional string specifying the pad method. The optional values are "zeros", "border" or "reflection". Default: `'zeros'` .
- **align_corners** (*bool, optional*) – If set to *True*, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to *False*, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. Default: `False` .

Returns

Tensor, dtype is the same as *input* and whose shape is (N, C, H_{out}, W_{out}) (4-D) and $(N, C, D_{out}, H_{out}, W_{out})$ (5-D).

Raises

- **TypeError** – If *input* or *grid* is not a Tensor.
- **TypeError** – If the dtypes of *input* and *grid* are inconsistent.
- **TypeError** – If the dtype of *input* or *grid* is not a valid type.
- **TypeError** – If *align_corners* is not a boolean value.
- **ValueError** – If the rank of *input* or *grid* is not equal to 4(4-D case) or 5(5-D case).
- **ValueError** – If the first dimension of *input* is not equal to that of *grid*.
- **ValueError** – If the last dimension of *grid* is not equal to 2(4-D case) or 3(5-D case).
- **ValueError** – If *mode* is not "bilinear", "nearest" or a string value.
- **ValueError** – If *padding_mode* is not "zeros", "border", "reflection" or a string value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.arange(16).reshape((2, 2, 2, 2)).astype(np.float32))
>>> grid = Tensor(np.arange(0.2, 1, 0.1).reshape((2, 2, 1, 2)).astype(np.float32))
>>> output = ops.grid_sample(input_x, grid, mode='bilinear', padding_mode='zeros',
...                          align_corners=True)
>>> print(output)
[[[[ 1.9       ]
   [ 2.1999998]]
  [[ 5.9       ]
   [ 6.2       ]]]
 [[[10.5       ]
   [10.8       ]]
  [[14.5       ]
   [14.8       ]]]]
```

mindspore.ops.interpolate

mindspore.ops.**interpolate**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*)

Samples the input Tensor to the given size or scale_factor by using one of the interpolate algorithms.

Parameters

- **input** (Tensor) –Tensor to be resized. Input tensor must be a 3-D, 4-D, or 5-D tensor with shape $(N, C, [optionalD], [optionalH], W)$, with data type of float.
- **size** (*Union[int, tuple[int], list[int]], optional*) –The target size. If size is a tuple or list, its length should be the same as the number of dimensions in input after removing the first two dimensions N, C. One and only one of size and scale_factor can be set to None. Default: None .
- **scale_factor** (*Union[float, tuple[float], list[float]], optional*) –The scale factor of new size of the tensor. If scale_factor is a tuple or list, its length should be the same as the number of dimensions in input after removing the first two dimensions N, C. One and only one of size and scale_factor can be set to None. Default: None .
- **mode** (*str, optional*) –The sampling algorithm. Default: "nearest" . One of the following sampling methods can be used:
  - 'nearest': the nearest neighbours interpolation.
  - 'linear': Linear interpolation, 3D only.
  - 'bilinear': Bilinear interpolation, 4D only.
  - 'trilinear': Trilinear interpolation, 5D only.
  - 'bicubic': Double trilinear interpolation, 4D only.
  - 'area': area interpolation.
  - 'nearest-exact': matches Scikit-Image and PIL nearest neighbours interpolation algorithms and fixes knows issues with *nearest*, for 3D and 4D.
- **align_corners** (*bool, optional*) –Whether to use corner alignment for coordinate mapping. Assuming a transformation is applied to the input Tensor along the x-axis, the specific calculation formula is as follows:

```
ori_i = new_length != 1 ? new_i * (ori_length - 1) / (new_length - 1) : 0   # 'align_
↪corners' = True

ori_i = new_length > 1 ? (new_i + 0.5) * ori_length / new_length - 0.5 : 0  # 'align_
↪corners' = False
```

  Among them, *ori_length* and *new_length* represent the length of the Tensor before and after transformation along the x-axis respectively; *new_i* represents the coordinate of the i-th element along the x-axis after transformation; *ori_i* represents the corresponding coordinate of the original data along the x-axis.

  This is only valid for 'linear', 'bilinear', or 'bicubic' modes. Default: False .
- **recompute_scale_factor** (*bool, optional*) –Recalculate *scale_factor*.
  - If True, the parameter *size* will be calculated using the value of the *scale_factor*, and finally scaled using the value of *size*.
  - If False, the value of *size* or *scale_factor* will be used for direct interpolation. Default: None .

Note: The 'nearest-exact' mode is the same as the nearest-neighbor interpolation algorithm used in scikit-image and PIL. The 'nearest' mode produces the same results as the INTER_NEAREST interpolation algorithm used in OpenCV.

Args Support List and Supported Platforms:

mode	input.dim	align_corners	scale_factor	device
nearest	3	-	×	Ascend,GPU,CPU
	4	-	×	Ascend,GPU,CPU
	5	-	√	Ascend,GPU,CPU
linear	3	√	×	Ascend,GPU,CPU
bilinear	4	√	×	Ascend,GPU,CPU
bicubic	4	√	×	Ascend,GPU,CPU
area	3	-	√	Ascend,GPU,CPU
	4	-	√	Ascend,GPU,CPU
	5	-	√	Ascend,GPU,CPU
nearest-exact	3	-	×	Ascend,CPU
	4	-	×	Ascend,CPU
trilinear	5	√	√	Ascend,GPU,CPU

- \- indicates that there is no such parameter.
- × indicates that this parameter is not currently supported.
- √ indicates that this parameter is supported.

Returns

Tensor, resized, whose dimensions and dtype are the same as *input*.

Raises

- **TypeError** – *input* is not a Tensor.
- **ValueError** – Both *size* and *scale_factor* are not empty.
- **ValueError** – Both *size* and *scale_factor* are empty.
- **ValueError** – When *size* is a tuple or list, its length is not equal to *input.ndim - 2*.
- **ValueError** – When *scale_factor* is a tuple or list, its length is not equal to *input.ndim - 2*.
- **ValueError** – *mode* is not in the list of supported modes.
- **ValueError** – *input.ndim* is not in the list of supported dimensions for the corresponding mode.
- **ValueError** – *size* is not empty, *recompute_scale_factor* is not empty.
- **ValueError** – *scale_factor* is not in the corresponding list of supported values.
- **ValueError** – *align_corners* is not in the corresponding list of supported values.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input = Tensor([[[1, 2, 3], [4, 5, 6]]], mindspore.float32)
>>> output = ops.interpolate(input, size=(6,), mode='nearest')
>>> print(output)
[[[1. 1. 2. 2. 3. 3.]
  [4. 4. 5. 5. 6. 6.]]]
```

mindspore.ops.iou

mindspore.ops.**iou**(*anchor_boxes*, *gt_boxes*, *mode='iou'*)

Calculates intersection over union for boxes.

Computes the intersection over union (IOU) or the intersection over foreground (IOF) based on the ground-truth and predicted regions.

$$\text{IOU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

$$\text{IOF} = \frac{\text{Area of Overlap}}{\text{Area of Ground Truth}}$$

Warning: In Ascend, only computation of float16 data is supported. To avoid overflow, the input length and width are scaled by 0.2 internally.

Parameters

- **anchor_boxes** (Tensor) – Anchor boxes, tensor of shape $(N, 4)$. N indicates the number of anchor boxes, and the value 4 refers to four boundary coordinates of the predicted area "x0", "y0", "x1", and "y1". Data type must be either float16, float32 or float64.
- **gt_boxes** (Tensor) – Ground truth boxes, tensor of shape $(M, 4)$. M indicates the number of ground truth boxes, and the value 4 refers to four boundary coordinates of the truth area "x0", "y0", "x1", and "y1". Data type must be either float16, float32 or float64.
- **mode** (*str*) – The mode is used to specify the calculation method, now supporting 'iou' (intersection over union) or 'iof' (intersection over foreground) mode. Default: 'iou' .

Returns

Tensor, the IOU/IOF values, tensor of shape (M, N) , with the same data type as *anchor_boxes*.

Raises

KeyError – When *mode* is not 'iou' or 'iof'.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> anchor_boxes = Tensor(np.random.randint(1.0, 5.0, [3, 4]), mindspore.float16)
>>> gt_boxes = Tensor(np.random.randint(1.0, 5.0, [3, 4]), mindspore.float16)
>>> mode = 'iou'
>>> output = ops.iou(anchor_boxes, gt_boxes, mode)
>>> print(output.shape)
(3, 3)
```

mindspore.ops.pad

mindspore.ops.**pad**(*input_x*, *padding*, *mode='constant'*, *value=None*)

Pads the input tensor according to the padding.

Parameters

- **input_x** (Tensor) –Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions which is required to be no more than 5 in Ascend.
- **padding** (*Union[tuple[int], list[int], Tensor]*) –Filling position of pad where the negative value is not supported while running in Ascend. $\left\lfloor \frac{\text{len(padding)}}{2} \right\rfloor$ dimensions of *input_x* will be padded.

  Example: to pad only the last dimension of the input tensor, then *padding* has the form (padding_left, padding_right);

  Example: to pad the last 2 dimensions of the input tensor, then use (padding_left, padding_right, padding_top, padding_bottom);

  Example: to pad the last 3 dimensions, use (padding_left, padding_right, padding_top, padding_bottom, padding_front, padding_ba
  and so on.
- **mode** (*str, optional*) –Pad filling mode, 'constant', 'reflect', 'replicate' or 'circular'. Default: 'constant'.

  For 'constant' mode, please refer to *mindspore.nn.ConstantPad1d* as an example to understand this filling pattern and extend the padding pattern to n dimensions.

  For 'reflect' mode, please refer to *mindspore.nn.ReflectionPad1d* as an example to understand this filling pattern. The reflect mode is used to pad the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.

  For 'replicate' mode, please refer to *mindspore.nn.ReplicationPad1d* as an example to understand this filling pattern. The replicate mode is used to pad the last three dimensions of 4D or 5D input, the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.

  For 'circular' mode, the pixels from one edge of the image are wrapped around to the opposite edge, such that the pixel on the right edge of the image is replaced with the pixel on the left edge, and the pixel on the bottom edge is replaced with the pixel on the top edge. The circular mode is used to pad the last three dimensions of 4D or 5D input, the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.
- **value** (*Union[int, float, None], optional*) –Valid only in 'constant' mode. Set the padding value in 'constant' mode. If the value is None, 0 is used as the default padding value. Default: None.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** –If *padding* is not an int of tuple or int of list.
- **TypeError** –If *input_x* is not a Tensor.
- **ValueError** –If length of *padding* is not even.
- **ValueError** –If length of *padding* is greater than 6.
- **ValueError** –If *mode* is not `'constant'` and *value* not `None`.
- **ValueError** –If rank of *input_x* is more than 5 while running in Ascend.
- **ValueError** –If *paddings* contains negative value while running in Ascend.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> import numpy as np
>>> x = ms.Tensor(np.arange(1 * 2 * 2 * 2).reshape((1, 2, 2, 2)), dtype=ms.float64)
>>> output = ops.pad(x, [1, 0, 0, 1], mode='constant', value=6.0)
>>> print(output)
[[[[6. 0. 1.]
   [6. 2. 3.]
   [6. 6. 6.]]
  [[6. 4. 5.]
   [6. 6. 7.]
   [6. 6. 6.]]]]
>>> output1 = ops.pad(x, (1, 0, 0, 1), mode='reflect')
>>> print(output1)
[[[[1. 0. 1.]
   [3. 2. 3.]
   [1. 0. 1.]]
  [[5. 4. 5.]
   [7. 6. 7.]
   [5. 4. 5.]]]]
>>> output2 = ops.pad(x, (1, 1, 2, 1), mode='replicate')
>>> print(output2)
[[[[0. 0. 1. 1.]
   [0. 0. 1. 1.]
   [0. 0. 1. 1.]
   [2. 2. 3. 3.]
   [2. 2. 3. 3.]]
  [[4. 4. 5. 5.]
   [4. 4. 5. 5.]
   [4. 4. 5. 5.]
   [6. 6. 7. 7.]
   [6. 6. 7. 7.]]]]
>>> output3 = ops.pad(x, (1, 1, 2, 1), mode='circular')
>>> print(output3)
[[[[1. 0. 1. 0.]
   [3. 2. 3. 2.]
   [1. 0. 1. 0.]
   [3. 2. 3. 2.]
   [1. 0. 1. 0.]]
```

(continues on next page)

(continued from previous page)

```
  [[5. 4. 5. 4.]
   [7. 6. 7. 6.]
   [5. 4. 5. 4.]
   [7. 6. 7. 6.]
   [5. 4. 5. 4.]]]]
```

mindspore.ops.padding

mindspore.ops.**padding**(*x*, *pad_dim_size=8*)

Extends the last dimension of the input tensor from 1 to pad_dim_size, by filling with 0.

Parameters

- **x** (Tensor) −The shape of tensor is $(x_1, x_2, ..., x_R)$. The rank of *x* must be at least 2. The last dimension of *x* must be 1. The data type is Number.
- **pad_dim_size** (*int*) −The value of the last dimension of *x* to be extended, which must be positive. Default: 8 .

Returns

Tensor, has the same type and shape as input shape value.

Raises

- **TypeError** −If *pad_dim_size* is not an int.
- **ValueError** −If *pad_dim_size* is less than 1.
- **ValueError** −If last dim of *x* is not equal to 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[8], [10]]), mindspore.float32)
>>> pad_dim_size = 4
>>> output = ops.padding(x, pad_dim_size)
>>> print(output)
[[ 8.  0.  0.  0.]
 [10.  0.  0.  0.]]
```

mindspore.ops.pixel_shuffle

mindspore.ops.**pixel_shuffle**(*input*, *upscale_factor*)

Applies the PixelShuffle operation over input *input* which implements sub-pixel convolutions with stride $1/r$. For more details, refer to Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network .

Typically, the *input* is of shape $(*, C \times r^2, H, W)$, and the output is of shape $(*, C, H \times r, W \times r)$, where r is an upscale factor and * is zero or more batch dimensions.

Parameters

- **input** (Tensor) –Tensor of shape $(*, C \times r^2, H, W)$. The dimension of *input* is larger than 2, and the length of third to last dimension can be divisible by *upscale_factor* squared.
- **upscale_factor** (*int*) –factor to shuffle the input Tensor, and is a positive integer. *upscale_factor* is the above-mentioned r.

Returns

- **output** (Tensor) - Tensor of shape $(*, C, H \times r, W \times r)$.

Raises

- **ValueError** –If *upscale_factor* is not a positive integer.
- **ValueError** –If the length of third to last dimension is not divisible by *upscale_factor* squared.
- **ValueError** –If the dimension of *input* is less than 3.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import ops
>>> input_x = np.arange(3 * 2 * 9 * 4 * 4).reshape((3, 2, 9, 4, 4))
>>> input_x = mindspore.Tensor(input_x, mindspore.dtype.int32)
>>> output = ops.pixel_shuffle(input_x, 3)
>>> print(output.shape)
(3, 2, 1, 12, 12)
```

mindspore.ops.pixel_unshuffle

mindspore.ops.**pixel_unshuffle**(*input*, *downscale_factor*)

Applies the PixelUnshuffle operation over input *input* which is the inverse of PixelShuffle. For more details, refer to Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network .

Typically, the input is of shape $(*, C, H \times r, W \times r)$, and the output is of shape $(*, C \times r^2, H, W)$, where r is a downscale factor and * is zero or more batch dimensions.

Parameters

- **input** (Tensor) –Tensor of shape $(*, C, H \times r, W \times r)$. The dimension of *input* is larger than 2, and the length of second to last dimension or last dimension can be divisible by *downscale_factor* .

- **downscale_factor** (*int*) –factor to unshuffle the input Tensor, and is a positive integer. *downscale_factor* is the above-mentioned r.

Returns

- **output** (Tensor) - Tensor of shape $(*, C \times r^2, H, W)$.

Raises

- **ValueError** –If *downscale_factor* is not a positive integer.
- **ValueError** –If the length of second to last dimension or last dimension is not divisible by *downscale_factor* .
- **ValueError** –If the dimension of *input* is less than 3.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = np.arange(8 * 8).reshape((1, 1, 8, 8))
>>> input_x = mindspore.Tensor(input_x, mindspore.dtype.int32)
>>> output = ops.pixel_unshuffle(input_x, 2)
>>> print(output.shape)
(1, 4, 4, 4)
```

mindspore.ops.rotary_position_embedding

mindspore.ops.**rotary_position_embedding**(*x*, *cos*, *sin*, *mode=0*)

Implements the Rotary Position Embedding algorithm. Refer to paper Enhanced Transformer with Rotary Position Embedding.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **x** (Tensor) –4D tensor, with float16, bfloat16 or float32 data type.
- **cos** (Tensor) –4D constant, has the same type as *x* , in range of [-1, 1].
- **sin** (Tensor) –Same with *cos* .
- **mode** (*int*) –An optional attribute. Used to select a calculation mode. 0: rotate_half(GPT-NeoX style); 1: rotate_interleaved(GPT-J style). Defaults to `0` .

Table 17: Config layout constraints

Args	RotateHalf(mode:0)	RotateInterleaved(mode:1)
x	Supported layout: 11SD, B1SD, BNSD; D < 896 and D is an Even. B, N < 1000;	Supported layout: 11SD, B1SD, BNSD; D < 896 and D is an Even. B, N < 1000;
cos	Support layout for different values of *x*: *x* is BNSD: 11SD, B1SD, BNSD; *x* is BSND: 1S1D, BS1D, BSND; *x* is SBND: S11D, SB1D, SBND	Support layout for different values of *x*: *x* is BNSD: 11SD; *x* is BSND: 1S1D; *x* is SBND: S11D
sin	Same with *cos* .	Same with *cos* .

Note: When the layout is BNSD, B * N > 8S and D is 32-bytes alignment, the performance is poor. Therefore, this interface cannot be called.

Returns

Tensor, has the same dtype and shape as the *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *cos* is not a Tensor.
- **TypeError** – If *sin* is not a Tensor.
- **TypeError** – If *mode* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.uniform(-2, 2, (4, 8192, 4, 128)))
>>> cos = Tensor(np.random.uniform(-1, 1, (1, 8192, 1, 128)))
>>> sin = Tensor(np.random.uniform(-1, 1, (1, 8192, 1, 128)))
>>> output = ops.rotary_position_embedding(x, cos, sin, 0)
>>> print(output.shape)
(4, 8192, 4, 128)
```

mindspore.ops.rotated_iou

mindspore.ops.**rotated_iou**(*boxes*, *query_boxes*, *trans=False*, *mode=0*, *is_cross=True*, *v_threshold=0.0*, *e_threshold=0.0*)

Calculate the overlap area between rotated rectangles.

> **Warning:** This is an experimental API that is subject to change or deletion.

Note: The input data types supported by the Ascend platform include bfloat16, float16, float32.

Parameters

- **boxes** (Tensor) –The first set of rectangles which has a shape of $(B, N, 5)$.
- **query_boxes** (Tensor) –The second set of rectangles which has a shape of $(B, K, 5)$.
- **trans** (*bool, optional*) –Distinguish the rectangles representations of boxes and query_boxes. If `True`, the format of boxes and query_boxes is `'xyxyt'`, else the format is `'xywht'`. The default value is `False`.
- **mode** (*int, optional*) –Distinguish the calculation mode. If the value is `1`, the calculation mode is `'iof'`, else the calculation mode is `'iou'`. The default value is `0`.
- **is_cross** (*bool, optional*) –If `True`, use cross-calculation, else use one-to-one calculation. The default value is `True`.
- **v_threshold** (*float, optional*) –Tolerance threshold for vertex determination. The default value is `0.0`.
- **e_threshold** (*float, optional*) –Tolerance threshold for edge intersection determination. The default value is `0.0`.

Returns

Tensor, the shape is (B, N, K).

Raises

- **TypeError** –If *boxes* is not a Tensor.
- **TypeError** –If *query_boxes* is not a Tensor.
- **ValueError** –If *boxes* and *query_boxes* do not has same first dim.
- **ValueError** –If the third dimension of *boxes* or *query_boxes* is not `5`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> a = np.random.uniform(0,1,(2,2,5)).astype(np.float16)
>>> b = np.random.uniform(0,1,(2,3,5)).astype(np.float16)
>>> box1 = Tensor(a)
>>> box2 = Tensor(b)
>>> output = ops.rotated_iou(box1, box2, trans=False, mode=0, is_cross=True)
```

mindspore.ops.upsample

mindspore.ops.**upsample**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*)

Alias for *mindspore.ops.interpolate()*.

Supported Platforms:

Ascend GPU CPU

2.4 Parameter Operation Functions

API Name	Description	Supported Platforms
mindspore.ops.assign	Assigns a parameter or tensor with a value.	Ascend GPU CPU
mindspore.ops.assign_add	Updates a parameter or tensor by adding a value to it.	Ascend GPU CPU
mindspore.ops.assign_sub	Updates a parameter or tensor by subtracting a value from it.	Ascend GPU CPU
mindspore.ops.scatter_add	Perform an addition update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_div	Perform a division update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_min	Perform a minimum update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_max	Perform a maximum update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_mul	Perform a multiplication update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_nd_add	Perform a sparse addition update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_nd_div	Perform a sparse division update on *input_x* based on the specified indices and update values.	GPU CPU
mindspore.ops.scatter_nd_max	Perform a sparse maximum update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_nd_min	Perform a sparse minimum update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_nd_mul	Perform a sparse multiplication update on *input_x* based on the specified indices and update values.	GPU CPU
mindspore.ops.scatter_nd_sub	Perform a sparse subtraction update on *input_x* based on the specified indices and update values.	Ascend GPU CPU
mindspore.ops.scatter_update	Updates the input tensor values using the given input indices and update values.	Ascend GPU CPU

2.4.1 mindspore.ops.assign

mindspore.ops.**assign**(*variable*, *value*)

Assigns a parameter or tensor with a value.

Support implicit type conversion and type promotion.

Parameters

- **variable** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **value** (Tensor) –The value to be assigned.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> value = mindspore.tensor([2.0], mindspore.float32)
>>> variable = mindspore.Parameter(mindspore.tensor([1.0], mindspore.float32), name="variable
↪")
>>> mindspore.ops.assign(variable, value)
>>> print(variable.asnumpy())
[2.]
```

2.4.2 mindspore.ops.assign_add

mindspore.ops.**assign_add**(*variable*, *value*)

Updates a parameter or tensor by adding a value to it.

Support implicit type conversion and type promotion.

Parameters

- **variable** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **value** (*Union[Tensor, Number]*) –The value to be added to the *variable*.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> variable = mindspore.Parameter([1], name="global_step")
>>> value = mindspore.tensor([100], dtype=mindspore.int32)
>>> mindspore.ops.assign_add(variable, value)
>>> print(variable.asnumpy())
[101]
```

2.4.3 mindspore.ops.assign_sub

mindspore.ops.**assign_sub**(*variable*, *value*)

Updates a parameter or tensor by subtracting a value from it.

Support implicit type conversion and type promotion.

Parameters

- **variable** (*Union[Parameter, Tensor]*) – The input parameter or tensor.
- **value** (Tensor) – The value to be subtracted from the *variable*.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> variable = mindspore.Parameter([1], name="global_step")
>>> value = mindspore.tensor([100], dtype=mindspore.int32)
>>> mindspore.ops.assign_sub(variable, value)
>>> print(variable.asnumpy())
[-99]
```

2.4.4 mindspore.ops.scatter_add

mindspore.ops.**scatter_add**(*input_x*, *indices*, *updates*)

Perform an addition update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j], :] \mathrel{+}= \text{updates}[i, ..., j, :]$$

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when *input_x* is of a low-precision data type.
- The shape of *updates* is *indices.shape* + *input_x.shape[1:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
...                               mindspore.float32), name="x")
>>> indices = mindspore.tensor([[0, 1], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_add(input_x, indices, updates)
>>> print(output)
[[ 1.  1.  1.]
 [19. 19. 19.]]
```

2.4.5 mindspore.ops.scatter_div

mindspore.ops.**scatter_div**(*input_x*, *indices*, *updates*)

Perform a division update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j], :] \ /= \text{updates}[i, ..., j, :]$$

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when *input_x* is of a low-precision data type.
- The shape of *updates* is *indices.shape* + *input_x.shape[1:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([[6.0, 6.0, 6.0], [2.0, 2.0, 2.0]],
...                               mindspore.float32), name="x")
>>> indices = mindspore.tensor([0, 1], mindspore.int32)
>>> updates = mindspore.tensor([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]], mindspore.float32)
>>> output = mindspore.ops.scatter_div(input_x, indices, updates)
>>> print(output)
[[3. 3. 3.]
 [1. 1. 1.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = mindspore.Parameter(mindspore.tensor([[105.0, 105.0, 105.0],
...                                                 [315.0, 315.0, 315.0]], mindspore.
 ↪float32), name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [1.0, 1.0, 1.0] = [105.0, 105.0, 105.0]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [3.0, 3.0, 3.0] = [105.0, 105.0, 105.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [105.0, 105.0, 105.0] / [5.0, 5.0, 5.0] = [21.0, 21.0, 21.0]
>>> # input_x[1] = [21.0, 21.0, 21.0] / [7.0, 7.0, 7.0] = [3.0, 3.0, 3.0]
>>> indices = mindspore.tensor([[0, 1], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[5.0, 5.0, 5.0], [7.0, 7.0, 7.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_div(input_x, indices, updates)
>>> print(output)
[[105. 105. 105.]
 [  3.   3.   3.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = mindspore.Parameter(mindspore.tensor([[105.0, 105.0, 105.0],
...                                                 [315.0, 315.0, 315.0]], mindspore.
 ↪float32), name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [3.0, 3.0, 3.0] = [35.0, 35.0, 35.0]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [1.0, 1.0, 1.0] = [315.0, 315.0, 315.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [5.0, 5.0, 5.0] = [63.0 63.0 63.0]
>>> # input_x[1] = [63.0 63.0 63.0] / [7.0, 7.0, 7.0] = [9.0, 9.0, 9.0]
>>> indices = mindspore.tensor([[1, 0], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[5.0, 5.0, 5.0], [7.0, 7.0, 7.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_div(input_x, indices, updates)
>>> print(output)
[[35. 35. 35.]
 [ 9.  9.  9.]]
```

2.4.6 mindspore.ops.scatter_min

mindspore.ops.**scatter_min**(*input_x*, *indices*, *updates*)

Perform a minimum update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \min(\text{input\_x}[\text{indices}[i, ..., j], :], \text{updates}[i, ..., j, :])$$

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when *input_x* is of a low-precision data type.
- The shape of *updates* is *indices.shape* + *input_x.shape[1:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.zeros((2, 3)),
...                               mindspore.float32), name="input_x")
>>> indices = mindspore.tensor([1, 0], mindspore.int32)
>>> update = mindspore.tensor(mindspore.ops.arange(0, 6).reshape((2, 3)), mindspore.float32)
>>> output = mindspore.ops.scatter_min(input_x, indices, update)
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]]
```

2.4.7 mindspore.ops.scatter_max

mindspore.ops.**scatter_max**(*input_x*, *indices*, *updates*)

Perform a maximum update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \max(\text{input\_x}[\text{indices}[i, ..., j], :], \text{updates}[i, ..., j, :])$$

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when *input_x* is of a low-precision data type.
- The shape of *updates* is *indices.shape* + *input_x.shape[1:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]],
...                               mindspore.float32), name="input_x")
>>> indices = mindspore.tensor([[0, 0], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor(mindspore.ops.ones([2, 2, 3]) * 88, mindspore.float32)
>>> output = mindspore.ops.scatter_max(input_x, indices, updates)
>>> print(output)
[[88. 88. 88.]
 [88. 88. 88.]]
```

2.4.8 mindspore.ops.scatter_mul

`mindspore.ops.`**`scatter_mul`**(*input_x*, *indices*, *updates*)

Perform a multiplication update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j], :] \mathrel{*}= \text{updates}[i, ..., j, :]$$

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when *input_x* is of a low-precision data type.
- The shape of *updates* is *indices.shape* + *input_x.shape[1:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.

- **indices** (Tensor) −The specified indices.
- **updates** (Tensor) −The update values.

Returns

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([[1.0, 1.0, 1.0],
...                               [2.0, 2.0, 2.0]], mindspore.float32), name="x")
>>> indices = mindspore.tensor([0, 1], mindspore.int32)
>>> updates = mindspore.tensor([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]], mindspore.float32)
>>> output = mindspore.ops.scatter_mul(input_x, indices, updates)
>>> print(output)
[[2. 2. 2.]
 [4. 4. 4.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = mindspore.Parameter(mindspore.tensor([[1.0, 1.0, 1.0],
...                                                 [2.0, 2.0, 2.0]], mindspore.float32),␣
 ↪name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [3.0, 3.0, 3.0] = [6.0, 6.0, 6.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [6.0, 6.0, 6.0] * [7.0, 7.0, 7.0] = [42.0, 42.0, 42.0]
>>> # input_x[1] = [42.0, 42.0, 42.0] * [9.0, 9.0, 9.0] = [378.0, 378.0, 378.0]
>>> indices = mindspore.tensor([[0, 1], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_mul(input_x, indices, updates)
>>> print(output)
[[  1.   1.   1.]
 [378. 378. 378.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = mindspore.Parameter(mindspore.tensor([[1.0, 1.0, 1.0],
...                                                 [2.0, 2.0, 2.0]], mindspore.float32),␣
 ↪name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [3.0, 3.0, 3.0] = [3.0, 3.0, 3.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [1.0, 1.0, 1.0] = [2.0, 2.0, 2.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [7.0, 7.0, 7.0] = [14.0, 14.0, 14.0]
>>> # input_x[1] = [14.0, 14.0, 14.0] * [9.0, 9.0, 9.0] = [126.0, 126.0, 126.0]
>>> indices = mindspore.tensor([[1, 0], [1, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_mul(input_x, indices, updates)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[  3.   3.   3.]
 [126. 126. 126.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = mindspore.Parameter(mindspore.tensor([[1.0, 1.0, 1.0],
...                                                 [2.0, 2.0, 2.0]], mindspore.float32),␣
 ↪name="x")
>>> # for indices = [[0, 1], [0, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [3.0, 3.0, 3.0] = [6.0, 6.0, 6.0]
>>> # step 2: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [7.0, 7.0, 7.0] = [7.0, 7.0, 7.0]
>>> # input_x[1] = [6.0, 6.0, 6.0] * [9.0, 9.0, 9.0] = [54.0, 54.0, 54.0]
>>> indices = mindspore.tensor([[0, 1], [0, 1]], mindspore.int32)
>>> updates = mindspore.tensor([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                             [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]], mindspore.float32)
>>> output = mindspore.ops.scatter_mul(input_x, indices, updates)
>>> print(output)
[[ 7.  7.  7.]
 [54. 54. 54.]]
```

2.4.9 mindspore.ops.scatter_nd_add

mindspore.ops.**scatter_nd_add**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse addition update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] \mathrel{+}= \text{updates}[i, ..., j]$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.
- **use_locking** (*bool, optional*) –Whether to protect the assignment by a lock. Default: False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([1, 2, 3, 4, 5, 6, 7, 8],
...                               mindspore.float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_add(input_x, indices, updates, False)
>>> print(output)
[ 1. 10.  9.  4. 12.  6.  7. 17.]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.zeros((4, 4, 4)), mindspore.
 ↪int32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
 ↪mindspore.int32)
>>> output = mindspore.ops.scatter_nd_add(input_x, indices, updates, False)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]]
```

2.4.10 mindspore.ops.scatter_nd_div

mindspore.ops.**scatter_nd_div**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse division update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] \ /= \text{updates}[i, ..., j]$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) – The input parameter or tensor.
- **indices** (Tensor) – The specified indices.

- **updates** (Tensor) −The update values.
- **use_locking** (*bool*) −Whether to protect the assignment by a lock. Default: `False` .

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([1, 2, 3, 4, 5, 6, 7, 8],
...                               mindspore.float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_div(input_x, indices, updates, False)
>>> print(output)
[1.         0.25       0.5        4.         0.71428573 6.
 7.         0.8888889 ]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.ones((4, 4, 4)), mindspore.
↪float32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
↪mindspore.float32)
>>> output = mindspore.ops.scatter_nd_div(input_x, indices, updates, False)
>>> print(output)
[[[1.         1.         1.         1.        ]
  [0.5        0.5        0.5        0.5       ]
  [0.33333334 0.33333334 0.33333334 0.33333334]
  [0.25       0.25       0.25       0.25      ]]
 [[1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]]
 [[0.2        0.2        0.2        0.2       ]
  [0.16666667 0.16666667 0.16666667 0.16666667]
  [0.14285715 0.14285715 0.14285715 0.14285715]
  [0.125      0.125      0.125      0.125     ]]
 [[1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]]]
```

2.4.11 mindspore.ops.scatter_nd_max

mindspore.ops.**scatter_nd_max**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse maximum update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] = \max(\text{input\_x}[\text{indices}[i, ..., j]], \text{updates}[i, ..., j])$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) – The input parameter or tensor.
- **indices** (Tensor) – The specified indices.
- **updates** (Tensor) – The update values.
- **use_locking** (*bool*) – Whether to protect the assignment by a lock. Default: False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([1, 2, 3, 4, 5, 6, 7, 8],
...                                mindspore.float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_max(input_x, indices, updates, False)
>>> print(output)
[1. 8. 6. 4. 7. 6. 7. 9.]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.ones((4, 4, 4)), mindspore.
↪int32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
↪mindspore.int32)
>>> output = mindspore.ops.scatter_nd_max(input_x, indices, updates, False)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[1 1 1 1]
```

(continues on next page)

(continued from previous page)

```
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
```

2.4.12 mindspore.ops.scatter_nd_min

mindspore.ops.**scatter_nd_min**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse minimum update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] = \min(\text{input\_x}[\text{indices}[i, ..., j]], \text{updates}[i, ..., j])$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) – The input parameter or tensor.
- **indices** (Tensor) – The specified indices.
- **updates** (Tensor) – The update values.
- **use_locking** (*bool*) – Whether to protect the assignment by a lock. Default: False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.ones(8) * 10, mindspore.
 ↪float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_min(input_x, indices, updates, False)
>>> print(output)
[10.  8.  6. 10.  7. 10. 10.  9.]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.ones((4, 4, 4)) * 10,␣
 ↪mindspore.int32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
 ↪mindspore.int32)
>>> output = mindspore.ops.scatter_nd_min(input_x, indices, updates, False)
>>> print(output)
[[[ 1  1  1  1]
  [ 2  2  2  2]
  [ 3  3  3  3]
  [ 4  4  4  4]]
 [[10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]]
 [[ 5  5  5  5]
  [ 6  6  6  6]
  [ 7  7  7  7]
  [ 8  8  8  8]]
 [[10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]]]
```

2.4.13 mindspore.ops.scatter_nd_mul

mindspore.ops.**scatter_nd_mul**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse multiplication update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] *= \text{updates}[i, ..., j]$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) – The input parameter or tensor.
- **indices** (Tensor) – The specified indices.

- **updates** (Tensor) −The update values.
- **use_locking** (*bool*) −Whether to protect the assignment by a lock. Default: False .

Returns

Tensor

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([1, 2, 3, 4, 5, 6, 7, 8],
...                               mindspore.float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_mul(input_x, indices, updates)
>>> print(output)
[ 1. 16. 18.  4. 35.  6.  7. 72.]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.ones((4, 4, 4)), mindspore.
 ↪int32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
 ↪mindspore.int32)
>>> output = mindspore.ops.scatter_nd_mul(input_x, indices, updates)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
```

2.4.14 mindspore.ops.scatter_nd_sub

mindspore.ops.**scatter_nd_sub**(*input_x*, *indices*, *updates*, *use_locking=False*)

Perform a sparse subtraction update on *input_x* based on the specified indices and update values.

$$\text{input\_x}[\text{indices}[i, ..., j]] \mathrel{-}= \text{updates}[i, ..., j]$$

Note:

- Support implicit type conversion and type promotion.
- The dimension of *indices* is at least 2, and its shape must be *indices.shape[-1] <= len(indices.shape)*.
- The shape of *updates* is *indices.shape[:-1] + input_x.shape[indices.shape[-1]:]*.

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –The specified indices.
- **updates** (Tensor) –The update values.
- **use_locking** (*bool*) –Whether to protect the assignment by a lock. Default: False .

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input_x = mindspore.Parameter(mindspore.tensor([1, 2, 3, 4, 5, 6, 7, 8],
...                                mindspore.float32), name="x")
>>> indices = mindspore.tensor([[2], [4], [1], [7]], mindspore.int32)
>>> updates = mindspore.tensor([6, 7, 8, 9], mindspore.float32)
>>> output = mindspore.ops.scatter_nd_sub(input_x, indices, updates, False)
>>> print(output)
[ 1. -6. -3.  4. -2.  6.  7. -1.]
>>> input_x = mindspore.Parameter(mindspore.tensor(mindspore.ops.zeros((4, 4, 4)), mindspore.
 ↪int32))
>>> indices = mindspore.tensor([[0], [2]], mindspore.int32)
>>> updates = mindspore.tensor([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                             [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]],␣
 ↪mindspore.int32)
>>> output = mindspore.ops.scatter_nd_sub(input_x, indices, updates, False)
>>> print(output)
[[[-1 -1 -1 -1]
  [-2 -2 -2 -2]
  [-3 -3 -3 -3]
  [-4 -4 -4 -4]]
 [[ 0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
  [ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]]
 [[-5 -5 -5 -5]
  [-6 -6 -6 -6]
  [-7 -7 -7 -7]
  [-8 -8 -8 -8]]
 [[ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]]]
```

2.4.15 mindspore.ops.scatter_update

mindspore.ops.**scatter_update**(*input_x*, *indices*, *updates*)

Updates the input tensor values using the given input indices and update values.

Note:

- Support implicit type conversion and type promotion.
- Since Parameter objects do not support type conversion, an exception will be thrown when input_x is of a low-precision data type.
- The *updates* with a shape of *indices.shape* + *input_x.shape[1:]* .

---

for each $i, \cdots, j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \text{updates}[i, ..., j, :]$$

Parameters

- **input_x** (*Union[Parameter, Tensor]*) –The input parameter or tensor.
- **indices** (Tensor) –Specify the indices for update operation. If there are duplicates in indices, the order for updating is undefined.
- **updates** (Tensor) –The values to update.

Returns

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> np_x = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> input_x = mindspore.Parameter(np_x, name="x")
>>> indices = mindspore.tensor([0, 1], mindspore.int32)
>>> np_updates =  mindspore.tensor([[2.0, 1.2, 1.0], [3.0, 1.2, 1.0]])
>>> updates =  mindspore.tensor(np_updates, mindspore.float32)
>>> output = mindspore.ops.scatter_update(input_x, indices, updates)
>>> print(output)
[[2. 1.2  1.]
 [3. 1.2  1.]]
```

2.5 Differential Functions

API Name	Description	Supported Platforms
mindspore.ops.derivative	This function is designed to calculate the higher order differentiation of given composite function.	Ascend GPU CPU
mindspore.ops.jet	This function is designed to calculate the higher order differentiation of given composite function.	Ascend GPU CPU
mindspore.ops.stop_gradient	StopGradient is used for eliminating the effect of a value on the gradient, such as truncating the gradient propagation from an output of a function.	Ascend GPU CPU

2.5.1 mindspore.ops.derivative

mindspore.ops.**derivative**(*fn*, *primals*, *order*)

This function is designed to calculate the higher order differentiation of given composite function. To figure out *order*-th order differentiations, original inputs and order must be provided together. In particular, the value of input first order derivative is set to 1, while the other to 0.

Note: If *primals* is tensor of int type, it will be converted to tensor of float type.

Parameters

- **fn** (*Union[Cell, function]*) –Function to do TaylorOperation.
- **primals** (*Union[Tensor, tuple[Tensor]]*) –The inputs to *fn*.
- **order** (*int*) –The order of differentiation.

Returns

Tuple(out_primals, out_series)

- **out_primals** (Union[Tensor, list[Tensor]]) - The output of *fn(primals)*.
- **out_series** (Union[Tensor, list[Tensor]]) - The *order*-th order of derivative of output with respect to the inputs.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> mindspore.set_context(mode=mindspore.GRAPH_MODE)
>>> class Net(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.sin = mindspore.ops.Sin()
...         self.exp = mindspore.ops.Exp()
...     def construct(self, x):
...         out1 = self.sin(x)
...         out2 = self.exp(out1)
...         return out2
>>>
>>> primals = mindspore.tensor([[1, 2], [3, 4]], mindspore.float32)
>>> order = 3
>>> net = Net()
>>> out_primals, out_series = mindspore.ops.derivative(net, primals, order)
>>> print(out_primals, out_series)
[[2.319777  2.4825778]
 [1.1515628 0.4691642]] [[-4.0515366   3.6724353 ]
 [ 0.5053504  -0.52061415]]
```

2.5.2 mindspore.ops.jet

mindspore.ops.**jet**(*fn*, *primals*, *series*)

This function is designed to calculate the higher order differentiation of given composite function. To figure out first to *n*-th order differentiations, original inputs and first to *n*-th order derivative of original inputs must be provided together. Generally, it is recommended to set the values of given first order derivative to 1, while the other to 0, which is like the derivative of origin input with respect to itself.

Note: If *primals* is tensor of int type, it will be converted to Tensor of float type.

Parameters

- **fn** (*Union[Cell, function]*) –Function to do TaylorOperation.
- **primals** (*Union[Tensor, tuple[Tensor]]*) –The inputs to *fn*.
- **series** (*Union[Tensor, tuple[Tensor]]*) –The original 1st to nth order derivatives of the input. The index *i* of the zeroth dimension of the tensor corresponds to the *i+1* -th order derivative of the output with respect to the input.

Returns

Tuple(out_primals, out_series)

- **out_primals** (Union[Tensor, list[Tensor]]) - The output of *fn(primals)*.
- **out_series** (Union[Tensor, list[Tensor]]) - The *1* to *i+1* -th order of derivative of output with respect to the inputs.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> mindspore.set_context(mode=mindspore.GRAPH_MODE)
>>> class Net(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.sin = mindspore.ops.Sin()
...         self.exp = mindspore.ops.Exp()
...     def construct(self, x):
...         out1 = self.sin(x)
...         out2 = self.exp(out1)
...         return out2
>>> primals = mindspore.tensor([[1, 2], [3, 4]], mindspore.float32)
>>> series = mindspore.tensor([[[1, 1], [1, 1]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]],␣
↪mindspore.float32)
>>> net = Net()
>>> out_primals, out_series = mindspore.ops.jet(net, primals, series)
>>> print(out_primals, out_series)
[[2.319777  2.4825778]
 [1.1515628 0.4691642]] [[[ 1.2533808  -1.0331168 ]
  [-1.1400385  -0.3066662 ]]
 [[-1.2748207  -1.8274734 ]
  [ 0.966121    0.55551505]]
 [[-4.0515366   3.6724353 ]
  [ 0.5053504  -0.52061415]]]
```

2.5.3 mindspore.ops.stop_gradient

mindspore.ops.**stop_gradient**(*value*)

StopGradient is used for eliminating the effect of a value on the gradient, such as truncating the gradient propagation from an output of a function. For more details, please refer to Stop Gradient.

Parameters

value (*Any*) – The value whose effect on the gradient to be eliminated.

Returns

The same as *value*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> def f1(x):
...     return x ** 2
>>> x = 3.0
>>> f1(x)
9.0
>>> mindspore.ops.grad(f1)(mindspore.tensor(x))
Tensor(shape=[], dtype=Float32, value= 6)
>>>
>>> # The same function with stop_gradient, return a zero gradient because x is effectively␣
↪treated as a constant.
>>> def f2(x):
...     return mindspore.ops.stop_gradient(x) ** 2
>>> f2(x)
9.0
>>> mindspore.ops.grad(f2)(mindspore.tensor(x))
Tensor(shape=[], dtype=Float32, value= 0)
```

2.6 Debugging Functions

API Name	Description	Supported Platforms
*mindspore.ops.print_*	Outputs the inputs to stdout.	Ascend GPU CPU
mindspore.ops.tensordump	Save tensor in npy format.	Ascend

2.6.1 mindspore.ops.print_

mindspore.ops.**print_**(**input_x*)

Outputs the inputs to stdout. The outputs are printed to screen by default. It can also be saved in a file by setting the parameter *print_file_path* in *context*. *mindspore.parse_print()* can be employed to reload the data. For more information, please refer to *mindspore.set_context()* and *mindspore.parse_print()*. In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when outputting big tensor or outputting tensor intensively.

Note: In pynative mode, please use python print function. In Ascend platform with graph mode, the bool, int and float would be converted into tensor to print, and str remains unchanged. This function is used for debugging.

Parameters

input_x (*Union[Tensor, bool, int, float, str, tuple, list]*) – The inputs of **print_**. Supports multiple inputs which are separated by ','.

Returns

Invalid value, should be ignored.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> x = mindspore.tensor(mindspore.ops.ones([2, 1], mindspore.int32))
>>> y = mindspore.tensor(mindspore.ops.ones([2, 2], mindspore.int32))
>>> result = mindspore.ops.print_('Print Tensor x and Tensor y:', x, y)
Print Tensor x and Tensor y:
Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [1]])
Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 1],
 [1, 1]])
```

2.6.2 mindspore.ops.tensordump

mindspore.ops.**tensordump**(*file_name*, *tensor*, *mode='out'*)

Save tensor in npy format.

Warning:

- The parameter *mode* will no longer support the value 'all'.

In Parallel situation, tensordump will dump slice of data at each rank. In Ascend platform with graph mode, Your code OpA -> OpB may compiled as OpA -> RedistributionOps -> OpB.

Note: The redistribution operator is introduced, Due to inter-device communication and shard strategies in the static graph parallel scenario.

In case of OpA -> OpB, the dump data of OpA's output is equal to OpB's input.

But in case of OpA -> RedistributionOps -> OpB, The dump data of OpA's output is not equal to OpB's input (Due to the redistribution operators). So the parameter mode is to handle this situation.

Assuming OpA's output is used as both tensordump's input parameter and OpB's input parameter. Different requirements of saving dump data can be achieved by configuring parameter *mode* :

- If the *mode* is 'out', the dump data contains only OpA's output slice.
- If the *mode* is 'in', the dump data contains only OpB's input slice.

For *mode* 'in', the input slice npy file format is: fileName_dumpMode_dtype_id.npy.

For *mode* 'out', the output slice npy file format is: fileName_dtype_id.npy.

- fileName: Value of the parameter file_name (if parameter *file_name* is a user-specified path, the value of fileName is the last level of the path).
- dumpMode: Value of the parameter *mode*.
- dtype: The original data type. Data of type bfloat16 stored in the .npy file will be converted to float32.
- id: An auto increment ID.

Note:

- In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when outputting big tensor or outputting tensor intensively.

- The operator of tensordump doesn't support in control flow.
- If current parallel mode is STAND_ALONE, *mode* should only be 'out'.
- This function is used for debugging.

Parameters

- **file_name** (*str*) −The path of the npy file saves.
- **tensor** (Tensor) −The tensor that user want to dump.
- **mode** (*str, optional*) −Used to control tensordump behavior, available value is one of ['in', 'out']. Default `out` .

Supported Platforms:

`Ascend`

Examples

Note: Using msrun command to run below example: msrun −worker_num=2 −local_worker_num=2 −master_port=11450 −log_dir=msrun_log −join=True −cluster_time_out=300 tensordump_example.py

---

```
>>> import os
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, context
>>> from mindspore.communication import init, get_rank
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.nn.utils import no_init_parameters
>>> init()
>>> rank_id = get_rank()
>>> dump_path = f'rank_{rank_id}_mul1_mul2.npy'
>>> class Net(nn.Cell):
...     def __init__(self, strategy1, strategy2):
...         super(Net, self).__init__()
...         self.matmul1 = mindspore.ops.MatMul().shard(strategy1)
...         self.matmul2 = mindspore.ops.MatMul().shard(strategy2)
...
...     def construct(self, x, y, b):
...         out1 = self.matmul1(x, y)
...         mindspore.ops.tensordump(dump_path, out1, 'out')
...         out2 = self.matmul2(out1, b)
...         return out2
...
>>> mindspore.set_context(mode=mindspore.GRAPH_MODE)
>>> os.environ["MS_DEV_SAVE_GRAPHS"] = "2"
>>> strategy1 = ((1, 2), (2, 1))
>>> strategy2 = ((1, 2), (2, 1))
>>> with no_init_parameters():
>>>     net = Net(strategy1, strategy2)
>>> x = mindspore.tensor(0.1 * mindspore.ops.randn(64, 64), mindspore.float32)
>>> y = mindspore.tensor(0.1 * mindspore.ops.randn(64, 64), mindspore.float32)
>>> b = mindspore.tensor(0.1 * mindspore.ops.randn(64, 64), mindspore.float32)
>>> parallel_net = Autoparallel(net, parallel_mode="semi_auto")
```

(continues on next page)

(continued from previous page)

```
>>> parallel_net.dataset_strategy(config="full_batch")
>>> out = parallel_net(x, y, b)
>>> print(f"out shape is: {out.shape}")
>>> # out shape is (64, 64)
>>> matmul1_output_slice = np.load(f'rank_{rank_id}_mul1_mul2_float32_0.npy')      # load␣
↪matmul1's output slice
>>> print(f"matmul1_output_slice is loaded, shape is: {matmul1_output_slice.shape}")
>>> # matmul1_output_slice is loaded, shape is: (64, 64)
```

2.7 Sparse Functions

Warning: These are experimental APIs that are subject to change or deletion.

API Name	Description	Supported Platforms
mindspore.ops.dense_to_sparse_coo	Convert a Tensor to COOTensor.	GPU
mindspore.ops.dense_to_sparse_csr	Convert a Tensor to CSRTensor.	GPU
mindspore.ops.csr_to_coo	Converts a CSRTensor to COOTensor.	Ascend GPU CPU

2.7.1 mindspore.ops.dense_to_sparse_coo

mindspore.ops.**dense_to_sparse_coo**(*tensor:* Tensor)

Convert a Tensor to COOTensor.

Note: Only 2-D tensor is supported for now.

Parameters

tensor (Tensor) – A dense tensor, must be 2-D.

Returns

COOTensor, a sparse representation of the original dense tensor, containing the following parts.

- indices (Tensor): 2-D integer tensor, indicates the positions of *values* of the dense tensor.
- values (Tensor): 1-D tensor, indicates the non-zero values of the dense tensor.
- shape (tuple(int)): the shape of the COOTensor, is the same as the original dense tensor.

Raises

- **TypeError** – If *tensor* is not a tensor.
- **ValueError** – If *tensor* is not 2-D.

Supported Platforms:

GPU

Examples

```
>>> from mindspore import Tensor, ops
>>> import mindspore as ms
>>> x = Tensor([[1, 0], [-5, 0]], ms.float32)
>>> output = ops.dense_to_sparse_coo(x)
>>> print(output.indices)
[[0 0]
[1 0]]
>>> print(output.values)
[ 1. -5.]
>>> print(output.shape)
(2, 2)
```

2.7.2 mindspore.ops.dense_to_sparse_csr

mindspore.ops.**dense_to_sparse_csr**(*tensor:* Tensor)

Convert a Tensor to CSRTensor.

Note: Only 2-D tensor is supported for now.

Parameters

tensor (Tensor) – A dense tensor, must be 2-D.

Returns

CSRTensor, a sparse representation of the original dense tensor, containing the following parts.

- indptr (Tensor): 1-D integer tensor, indicates the start and end point for *values* in each row.
- indices (Tensor): 1-D integer tensor, indicates the column positions of all non-zero values of the input.
- values (Tensor): 1-D tensor, indicates the non-zero values of the dense tensor.
- shape (tuple(int)): the shape of the CSRTensor, is the same as the original dense tensor.

Raises

- **TypeError** – If input is not a tensor.
- **ValueError** – If input tensor is not 2-D.

Supported Platforms:

GPU

Examples

```
>>> from mindspore import Tensor, ops
>>> import mindspore as ms
>>> x = Tensor([[1, 0], [-5, 0]], ms.float32)
>>> output = ops.dense_to_sparse_csr(x)
>>> print(output.indptr)
[0 1 2]
>>> print(output.indices)
[0 0]
>>> print(output.shape)
(2, 2)
```

2.7.3 mindspore.ops.csr_to_coo

mindspore.ops.**csr_to_coo**(*tensor:* CSRTensor)

Converts a CSRTensor to COOTensor.

Note: Only 2-D CSRTensor is supported for now.

Parameters

tensor (CSRTensor) – A CSRTensor, must be 2-D.

Returns

2D COOTensor, the input tensor stored in COO format.

Raises

- **TypeError** – If input is not a COOTensor.
- **ValueError** – If input tensor is not 2-D.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> indptr = Tensor([0, 1, 2]).astype("int32")
>>> indices = Tensor([0, 1]).astype("int32")
>>> values = Tensor([2, 1]).astype("float32")
>>> shape = (2, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_to_coo(x)
>>> print(output.indices)
[[0 0]
[1 1]]
```

2.7.4 COO Functions

Warning: These are experimental APIs that are subject to change or deletion.

API Name	Description	Supported Platforms
mindspore.ops.coo_abs	Returns coo_absolute value of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_acos	Computes arccosine of input coo_tensors element-wise.	Ascend GPU CPU
mindspore.ops.coo_acosh	Computes inverse hyperbolic cosine of the inputs element-wise.	Ascend GPU CPU
mindspore.ops.coo_add	Computes the sum of x1(COOTensor) and x2(COOTensor), and return a new COOTensor based on the computed result and *thresh*.	GPU CPU
mindspore.ops.coo_asin	Computes arcsine of input coo_tensors element-wise.	Ascend GPU CPU
mindspore.ops.coo_asinh	Computes inverse hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.coo_atan	Computes the trigonometric inverse tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.coo_atanh	Computes inverse hyperbolic tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.coo_ceil	Rounds a COOTensor up to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.coo_concat	concatenates the input SparseTensor(COO format) along the specified dimension.	CPU
mindspore.ops.coo_cos	Computes cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.coo_cosh	Computes hyperbolic cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.coo_exp	Returns the element-wise exponential of a COOTensor.	Ascend GPU CPU
mindspore.ops.coo_expm1	Returns exponential then minus 1 of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_floor	Rounds a COOTensor down to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.coo_inv	Computes Reciprocal of input COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_isfinite	Determines which elements are finite for each position.	Ascend GPU CPU
mindspore.ops.coo_isinf	Determines which elements are inf or -inf for each position.	GPU CPU
mindspore.ops.coo_isnan	Determines which elements are NaN for each position.	Ascend GPU CPU
mindspore.ops.coo_log	Returns the natural logarithm of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_log1p	Returns the natural logarithm of one plus the input COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_neg	Returns a COOTensor with coo_negative values of the input COOTensor element-wise.	Ascend GPU CPU

continues on next page

Table 22 – continued from previous page

mindspore.ops.coo_relu	Computes ReLU (Rectified Linear Unit activation function) of input coo_tensors element-wise.	Ascend GPU CPU
mindspore.ops.coo_relu6	Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input coo_tensors element-wise.	Ascend GPU CPU
mindspore.ops.coo_round	Returns half to even of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_sigmoid	Sigmoid activation function.	Ascend GPU CPU
mindspore.ops.coo_sin	Computes sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.coo_sinh	Computes hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.coo_softsign	Softsign activation function.	Ascend GPU CPU
mindspore.ops.coo_sqrt	Computes sqrt of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_square	Returns square of a COOTensor element-wise.	Ascend GPU CPU
mindspore.ops.coo_tan	Computes tangent of *x* element-wise.	Ascend GPU CPU
mindspore.ops.coo_tanh	Computes hyperbolic tangent of input element-wise.	Ascend GPU CPU

mindspore.ops.coo_abs

mindspore.ops.**coo_abs**(*x:* COOTensor)

Returns coo_absolute value of a COOTensor element-wise.

$$out_i = |x_i|$$

Parameters

x (COOTensor) – The input COOTensor.

Returns

COOTensor, has the same shape as the *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_abs(x)
>>> print(output.values)
[1. 2.]
```

mindspore.ops.coo_acos

mindspore.ops.**coo_acos**(*x:* COOTensor)

Computes arccosine of input coo_tensors element-wise.

$$out_i = \cos^{-1}(x_i)$$

Parameters

x (COOTensor) – Input COOTensor.

Returns

COOTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_acos(x)
>>> print(output.values)
[3.1415927       nan]
```

mindspore.ops.coo_acosh

mindspore.ops.**coo_acosh**(*x:* COOTensor)

Computes inverse hyperbolic cosine of the inputs element-wise.

$$y_i = \cosh^{-1}(x_i)$$

Warning: Given an input COOTensor x, the function computes inverse hyperbolic cosine of every element. Input range is [1, inf].

Parameters

x (COOTensor) – The input COOTensor of inverse hyperbolic cosine function.

Returns

COOTensor, has the same shape and type as *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_acosh(x)
>>> print(output.values)
[     nan 1.316958]
```

mindspore.ops.coo_add

`mindspore.ops.`**`coo_add`**(*x1:* COOTensor, *x2:* COOTensor, *thresh:* Tensor)

Computes the sum of x1(COOTensor) and x2(COOTensor), and return a new COOTensor based on the computed result and *thresh*.

Parameters

- **x1** (COOTensor) – the first COOTensor to sum.
- **x2** (COOTensor) – the second COOTensor to sum.
- **thresh** (Tensor) – A 0-D Tensor, represents the magnitude threshold that determines if an output value/index pair take place. Its dtype should match that of the values if they are real. If output's value is less than the *thresh*, it will vanish.

Returns

A COOTensor, the result of sum.

Raises

- **ValueError** – If any input(x1/x2)'s indices's dim is not equal to 2.
- **ValueError** – If any input(x1/x2)'s values's dim is not equal to 1.
- **ValueError** – If any input(x1/x2)'s shape's dim is not equal to 1.
- **ValueError** – If thresh's dim is not equal to 0.
- **TypeError** – If any input(x1/x2)'s indices's type is not equal to int64.
- **TypeError** – If any input(x1/x2)'s shape's type is not equal to int64.
- **ValueError** – If any input(x1/x2)'s indices's length is not equal to its values's length.
- **TypeError** – If any input(x1/x2)'s values's type is not equal to anf of (int8/int16/int32/int64/float32/float64/complex64/complex128).
- **TypeError** – If thresh's type is not equal to anf of (int8/int16/int32/int64/float32/float64).
- **TypeError** – If x1's indices's type is not equal to x2's indices's type.
- **TypeError** – If x1's values's type is not equal to x2's values's type.

- **TypeError** –If x1's shape's type is not equal to x2's shape's type.
- **TypeError** –If (x1/x2)'s value's type is not matched with thresh's type.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import Tensor, COOTensor
>>> from mindspore import dtype as mstype
>>> from mindspore import context
>>> from mindspore import ops
>>> indics0 = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values0 = Tensor([1, 2], dtype=mstype.int32)
>>> shape0 = (3, 4)
>>> input0 = COOTensor(indics0, values0, shape0)
>>> indics1 = Tensor([[0, 0], [1, 1]], dtype=mstype.int64)
>>> values1 = Tensor([3, 4], dtype=mstype.int32)
>>> shape1 = (3, 4)
>>> input1 = COOTensor(indics1, values1, shape1)
>>> thres = Tensor(0, dtype=mstype.int32)
>>> out = ops.coo_add(input0, input1, thres)
>>> print(out)
COOTensor(shape=[3, 4], dtype=Int32, indices=Tensor(shape=[4, 2], dtype=Int64, value=
[[0 0]
 [0 1]
 [1 1]
 [1 2]]), values=Tensor(shape=[4], dtype=Int32, value=[3 1 4 2]))
```

mindspore.ops.coo_asin

mindspore.ops.**coo_asin**(*x:* COOTensor)

Computes arcsine of input coo_tensors element-wise.

$$out_i = \sin^{-1}(x_i)$$

Parameters

x (COOTensor) –Input COOTensor. The shape of COOTensor is $(N, *)$, where $*$ means any number of additional dimensions. The data type should be one of the following types: float16, float32, float64, complex64, complex128.

Returns

COOTensor, has the same shape and dtype as x.

Raises

- **TypeError** –If x is not a COOTensor.
- **TypeError** –If dtype of x is not float16, float32, float64, complex64, complex128.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_asin(x)
>>> print(output.values)
[-1.5707964        nan]
```

mindspore.ops.coo_asinh

mindspore.ops.**coo_asinh**(*x:* COOTensor)

Computes inverse hyperbolic sine of the input element-wise.

$$out_i = \sinh^{-1}(input_i)$$

Parameters

x (COOTensor) – The input COOTensor of inverse hyperbolic sine function.

Returns

COOTensor, has the same shape and type as *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_asinh(x)
>>> print(output.values)
[-0.8813736  1.4436355]
```

mindspore.ops.coo_atan

mindspore.ops.**coo_atan**(*x:* COOTensor)

Computes the trigonometric inverse tangent of the input element-wise.

$$out_i = \tan^{-1}(x_i)$$

Parameters

x (COOTensor) –The data type should be one of the following types: float16, float32.

Returns

A COOTensor, has the same type as the input.

Raises

- **TypeError** –If *x* is not a COOTensor.
- **TypeError** –If dtype of *x* is not float16 or float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_atan(x)
>>> print(output.values)
[-0.7853982  1.1071488]
```

mindspore.ops.coo_atanh

mindspore.ops.**coo_atanh**(*x:* COOTensor)

Computes inverse hyperbolic tangent of the input element-wise.

$$out_i = \tanh^{-1}(x_i)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

x (COOTensor) –Input COOTensor. The data type should be one of the following types: float16, float32.

Returns

A COOTensor, has the same type as the input.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is not float16 or float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_atanh(x)
>>> print(output.values)
[-inf  nan]
```

mindspore.ops.coo_ceil

mindspore.ops.**coo_ceil**(*x:* COOTensor)

Rounds a COOTensor up to the closest integer element-wise.

$$out_i = \lceil x_i \rceil = \lfloor x_i \rfloor + 1$$

Parameters

x (COOTensor) – The input COOTensor with a dtype of float16 or float32.

Returns

COOTensor, has the same shape as the *x*.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is not float16 or float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_ceil(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.coo_concat

mindspore.ops.**coo_concat**(*sp_input*, *concat_dim=0*)

concatenates the input SparseTensor(COO format) along the specified dimension.

> **Warning:** This is an experimental API that is subjected to change or deletion. Only supported on CPU now.

Parameters

- **sp_input** (*Union[list(COOTensor), tuple(COOTensor)]*) – for COOTensor input.
- **concat_dim** (*scalar*) – decide the dimension to concatenation along. The value must be in range [-rank, rank), where rank is the number of dimensions in each input SparseTensor. Default is 0.

Returns

- **output** (COOtensor) - the result of concatenates the input SparseTensor along the specified dimension. OutShape: OutShape[non concat_dim] is equal to InShape[non concat_dim] and OutShape[concat_dim] is all input concat_dim axis shape accumulate.

Raises

- **ValueError** – If only one sparse tensor input.
- **ValueError** – If Input COOTensor shape dim > 3. COOtensor shape dim size must be 2 now.

Supported Platforms:

CPU

Examples

```
>>> from mindspore import Tensor, ops, COOTensor
>>> from mindspore import dtype as mstype
>>> indices0 = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values0 = Tensor([1, 2], dtype=mstype.int32)
>>> shape0 = (3, 4)
>>> input0 = COOTensor(indices0, values0, shape0)
>>> indices1 = Tensor([[0, 0], [1, 1]], dtype=mstype.int64)
>>> values1 = Tensor([3, 4], dtype=mstype.int32)
>>> shape1 = (3, 4)
>>> input1 = COOTensor(indices1, values1, shape1)
>>> concat_dim = 1
>>> out = ops.coo_concat((input0, input1), concat_dim)
>>> print(out)
COOTensor(shape=[3, 8], dtype=Int32, indices=Tensor(shape=[4, 2], dtype=Int64, value=
[[0 1]
 [0 4]
 [1 2]
 [1 5]]), values=Tensor(shape=[4], dtype=Int32, value=[1 3 2 4]))
```

mindspore.ops.coo_cos

mindspore.ops.**coo_cos**(*x:* COOTensor)

Computes cosine of input element-wise.

$$out_i = \cos(x_i)$$

Warning: If use float64, there may be a problem of missing precision.

Parameters

x (COOTensor) – Input COOTensor.

Returns

COOTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64, complex64, complex128.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_cos(x)
>>> print(output.values)
[ 0.5403023  -0.41614684]
```

mindspore.ops.coo_cosh

mindspore.ops.**coo_cosh**(*x:* COOTensor)

Computes hyperbolic cosine of input element-wise.

$$out_i = \cosh(x_i)$$

Parameters

x (COOTensor) – The input COOTensor of hyperbolic cosine function, its data type must be float16, float32, float64, complex64 or complex128.

Returns

COOTensor, has the same shape as *x*.

Raises

- **TypeError** – If the dtype of *x* is not one of the following types: float16, float32, float64, complex64, complex128.
- **TypeError** – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_cosh(x)
>>> print(output.values)
[1.5430807 3.7621956]
```

mindspore.ops.coo_exp

mindspore.ops.**coo_exp**(*x:* COOTensor)

Returns the element-wise exponential of a COOTensor.

$$out_i = e^{x_i}$$

Parameters

x (COOTensor) – The input COOTensor.

Returns

COOTensor, has the same shape and dtype as the *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_exp(x)
>>> print(output.values)
[0.36787948 7.3890557 ]
```

mindspore.ops.coo_expm1

mindspore.ops.**coo_expm1**(*x:* COOTensor)

Returns exponential then minus 1 of a COOTensor element-wise.

$$out_i = e^{x_i} - 1$$

Parameters

x (COOTensor) –The input COOTensor with a dtype of float16 or float32.

Returns

COOTensor, has the same shape as the *x*.

Raises

- **TypeError** –If *x* is not a COOTensor.
- **TypeError** –If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_expm1(x)
>>> print(output.values)
[-0.63212055  6.389056  ]
```

mindspore.ops.coo_floor

mindspore.ops.**coo_floor**(*x:* COOTensor)

Rounds a COOTensor down to the closest integer element-wise.

$$out_i = \lfloor x_i \rfloor$$

Parameters

x (COOTensor) –The input COOTensor, its data type must be float16, float32 or float64.

Returns

COOTensor, has the same shape as *x*.

Raises

- **TypeError** –If *x* is not a COOTensor.
- **TypeError** –If dtype of *x* is not in [float16, float32, float64].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_floor(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.coo_inv

mindspore.ops.**coo_inv**(*x:* COOTensor)

Computes Reciprocal of input COOTensor element-wise.

$$out_i = \frac{1}{x_i}$$

Parameters

x (COOTensor) – Input COOTensor. Must be one of the following types: float16, float32 or int32.

Returns

COOTensor, has the same type and shape as input shape value.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is not one of float16, float32, int32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_inv(x)
>>> print(output.values)
[-1.   0.5]
```

mindspore.ops.coo_isfinite

mindspore.ops.**coo_isfinite**(*x:* COOTensor)

Determines which elements are finite for each position.

$$out_i = \begin{cases} \text{if } x_i = \text{Finite}, & True \\ \text{if } x_i \neq \text{Finite}, & False \end{cases}$$

Parameters

x (COOTensor) –The input COOTensor.

Returns

COOTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError –If x is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_isfinite(x)
>>> print(output.values)
[ True  True]
```

mindspore.ops.coo_isinf

mindspore.ops.**coo_isinf**(*x:* COOTensor)

Determines which elements are inf or -inf for each position.

$$out_i = \begin{cases} \text{if } x_i = \text{Inf}, & True \\ \text{if } x_i \neq \text{Inf}, & False \end{cases}$$

where Inf means infinitity or negative infinitity.

Parameters

x (COOTensor) –The input COOTensor.

Returns

COOTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError –If x is not a COOTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_isinf(x)
>>> print(output.values)
[False False]
```

mindspore.ops.coo_isnan

mindspore.ops.**coo_isnan**(*x:* COOTensor)

Determines which elements are NaN for each position.

$$out_i = \begin{cases} True, & \text{if } x_i = \text{Nan} \\ False, & \text{if } x_i \neq \text{Nan} \end{cases}$$

where Nan means not a number.

Parameters

x (COOTensor) – The input COOTensor.

Returns

COOTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError – If x is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_isnan(x)
>>> print(output.values)
[False False]
```

mindspore.ops.coo_log

mindspore.ops.**coo_log**(*x:* COOTensor)

Returns the natural logarithm of a COOTensor element-wise.

$$y_i = \log_e(x_i)$$

Warning: If the input value of operator Log is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

x (COOTensor) –The value must be greater than 0.

Returns

COOTensor, has the same shape and dtype as the *x*.

Raises

- **TypeError** –If *x* is not a COOTensor.
- **TypeError** –If dtype of *x* is not float16, float32 or float64 on GPU and CPU.
- **TypeError** –If dtype of *x* is not float16 or float32 on Ascend.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_log(x)
>>> print(output.values)
[       nan 0.69314575]
```

mindspore.ops.coo_log1p

mindspore.ops.**coo_log1p**(*x:* COOTensor)

Returns the natural logarithm of one plus the input COOTensor element-wise.

$$out_i = \log\_e(x_i + 1)$$

Parameters

x (COOTensor) –The input COOTensor, should have dtype of float16 or float32 and its value should be greater than -1.

Returns

COOTensor, has the same shape as the *x*.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_log1p(x)
>>> print(output.values)
[    -inf 1.0986123]
```

mindspore.ops.coo_neg

`mindspore.ops.coo_neg`(*x:* COOTensor)

Returns a COOTensor with coo_negative values of the input COOTensor element-wise.

$$out_i = -x_i$$

Parameters

x (`COOTensor`) – The input COOTensor with a dtype of Number.

Returns

COOTensor, has the same shape and dtype as input *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_neg(x)
>>> print(output.values)
[ 1. -2.]
```

mindspore.ops.coo_relu

mindspore.ops.**coo_relu**(*x:* COOTensor)

Computes ReLU (Rectified Linear Unit activation function) of input coo_tensors element-wise.

It returns $\max(x,\ 0)$ element-wise. Specially, the neurons with the negative output will be suppressed and the active neurons will stay the same.

$$ReLU(x) = (x)^+ = \max(0, x)$$

Parameters

x (COOTensor) –Input COOTensor with shape $(N, *)$, where $*$ means any number of additional dimensions. Its dtype is number.

Returns

COOTensor, has the same shape and dtype as the *x*.

Raises

- **TypeError** –If dtype of *x* is not a number.
- **TypeError** –If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_relu(x)
>>> print(output.values)
[0. 2.]
```

mindspore.ops.coo_relu6

mindspore.ops.**coo_relu6**(*x:* COOTensor)

Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input coo_tensors element-wise.

$$\text{ReLU6}(x) = \min(\max(0, x), 6)$$

It returns $\min(\max(0, x), 6)$ element-wise.

Parameters

x (COOTensor) –Input COOTensor, with float16 or float32 data type.

Returns

COOTensor, with the same dtype and shape as the *x*.

Raises

- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **TypeError** – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_relu6(x)
>>> print(output.values)
[0. 2.]
```

mindspore.ops.coo_round

mindspore.ops.**coo_round**(*x:* COOTensor)

Returns half to even of a COOTensor element-wise.

$$out_i \approx x_i$$

Parameters

x (COOTensor) – The input COOTensor.

Returns

COOTensor, has the same shape and type as the *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_round(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.coo_sigmoid

mindspore.ops.**coo_sigmoid**(*x:* COOTensor)

Sigmoid activation function.

Computes Sigmoid of input element-wise. The Sigmoid function is defined as:

$$\text{coo\_sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)}$$

where x_i is an element of the x.

Parameters

x (COOTensor) –Input COOTensor, the data type is float16, float32, float64, complex64 or complex128.

Returns

COOTensor, with the same type and shape as the x.

Raises

- **TypeError** –If dtype of *x* is not float16, float32, float64, complex64 or complex128.
- **TypeError** –If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_sigmoid(x)
>>> print(output.values)
[0.26894143 0.8807971 ]
```

mindspore.ops.coo_sin

mindspore.ops.**coo_sin**(*x:* COOTensor)

Computes sine of the input element-wise.

$$out_i = \sin(x_i)$$

Parameters

x (COOTensor) –Input COOTensor.

Returns

COOTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** –If *x* is not a COOTensor.

- **TypeError** – If dtype of *x* is not float16, float32 or float64, complex64, complex128.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_sin(x)
>>> print(output.values)
[-0.84147096  0.9092974 ]
```

mindspore.ops.coo_sinh

mindspore.ops.**coo_sinh**(*x:* COOTensor)

Computes hyperbolic sine of the input element-wise.

$$out_i = \sinh(x_i)$$

Parameters

x (COOTensor) – The input COOTensor of hyperbolic sine function.

Returns

COOTensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_sinh(x)
>>> print(output.values)
[-1.1752012  3.6268604]
```

mindspore.ops.coo_softsign

mindspore.ops.**coo_softsign**(*x:* COOTensor)

Softsign activation function.

The function is shown as follows:

$$\text{SoftSign}(x) = \frac{x}{1 + |x|}$$

Parameters

x (COOTensor) – Input COOTensor, with float16 or float32 data type.

Returns

COOTensor, with the same type and shape as the *x*.

Raises

- **TypeError** – If *x* is not a COOTensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_softsign(x)
>>> print(output.values)
[-0.5        0.6666667]
```

mindspore.ops.coo_sqrt

mindspore.ops.**coo_sqrt**(*x:* COOTensor)

Computes sqrt of a COOTensor element-wise.

$$out_i = \sqrt{x_i}$$

Parameters

x (COOTensor) – The input COOTensor with a dtype of Number.

Returns

COOTensor, has the same shape and dtype as the *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_sqrt(x)
>>> print(output.values)
[      nan 1.4142135]
```

mindspore.ops.coo_square

mindspore.ops.**coo_square**(*x:* COOTensor)

Returns square of a COOTensor element-wise.

$$out_i = (x_i)^2$$

Parameters

x (COOTensor) – The input COOTensor with a dtype of Number.

Returns

COOTensor, has the same shape and dtype as the *x*.

Raises

TypeError – If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_square(x)
>>> print(output.values)
[1. 4.]
```

mindspore.ops.coo_tan

mindspore.ops.**coo_tan**(*x:* COOTensor)

Computes tangent of *x* element-wise.

$$out_i = \tan(x_i)$$

Parameters

x (COOTensor) –The input COOTensor.

Returns

COOTensor, has the same shape as *x*.

Raises

TypeError –If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_tan(x)
>>> print(output.values)
[-1.5574077 -2.1850398]
```

mindspore.ops.coo_tanh

mindspore.ops.**coo_tanh**(*x:* COOTensor)

Computes hyperbolic tangent of input element-wise. The Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input COOTensor.

Parameters

x (COOTensor) –Input COOTensor, with float16 or float32 data type.

Returns

COOTensor, with the same type and shape as the *x*.

Raises

- **TypeError** –If dtype of *x* is neither float16 nor float32.
- **TypeError** –If *x* is not a COOTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, COOTensor
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mstype.int64)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = COOTensor(indices, values, shape)
>>> output = ops.coo_tanh(x)
>>> print(output.values)
[-0.7615942  0.9640276]
```

2.7.5 CSR Functions

Warning: These are experimental APIs that are subject to change or deletion.

API Name	Description	Supported Platforms
mindspore.ops.csr_abs	Returns csr_absolute value of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_acos	Computes arccosine of input csr_tensors element-wise.	Ascend GPU CPU
mindspore.ops.csr_acosh	Computes inverse hyperbolic cosine of the inputs element-wise.	Ascend GPU CPU
mindspore.ops.csr_add	Computes the linear combination of two input CSRTensors a and b.	GPU CPU
mindspore.ops.csr_asin	Computes arcsine of input csr_tensors element-wise.	Ascend GPU CPU
mindspore.ops.csr_asinh	Computes inverse hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.csr_atan	Computes the trigonometric inverse tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.csr_atanh	Computes inverse hyperbolic tangent of the input element-wise.	Ascend GPU CPU
mindspore.ops.csr_ceil	Rounds a CSRTensor up to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.csr_cos	Computes cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.csr_cosh	Computes hyperbolic cosine of input element-wise.	Ascend GPU CPU
mindspore.ops.csr_exp	Returns csr_exponential of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_expm1	Returns exponential then minus 1 of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_floor	Rounds a CSRTensor down to the closest integer element-wise.	Ascend GPU CPU
mindspore.ops.csr_inv	Computes Reciprocal of input CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_isfinite	Determines which elements are finite for each position.	Ascend GPU CPU
mindspore.ops.csr_isinf	Determines which elements are inf or -inf for each position.	GPU CPU

continues on next page

Table 23 – continued from previous page

mindspore.ops.csr_isnan	Determines which elements are NaN for each position.	Ascend GPU CPU
mindspore.ops.csr_log	Returns the natural logarithm of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_log1p	Returns the natural logarithm of one plus the input CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_mm	Return the matrix multiplication result of the right-multiply matrix (dense or CSRTensor) of the CSRTensor.	GPU
mindspore.ops.csr_neg	Returns a CSRTensor with csr_negative values of the input CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_relu	Computes ReLU (Rectified Linear Unit activation function) of input csr_tensors element-wise.	Ascend GPU CPU
mindspore.ops.csr_relu6	Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input csr_tensors element-wise.	Ascend GPU CPU
mindspore.ops.csr_round	Returns half to even of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_sigmoid	Sigmoid activation function.	Ascend GPU CPU
mindspore.ops.csr_sin	Computes sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.csr_sinh	Computes hyperbolic sine of the input element-wise.	Ascend GPU CPU
mindspore.ops.csr_softmax	Calculates the softmax of a CSRTensorMatrix.	GPU CPU
mindspore.ops.csr_softsign	Softsign activation function.	Ascend GPU CPU
mindspore.ops.csr_sqrt	Returns sqrt of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_square	Returns square of a CSRTensor element-wise.	Ascend GPU CPU
mindspore.ops.csr_tan	Computes tangent of x element-wise.	Ascend GPU CPU
mindspore.ops.csr_tanh	Computes hyperbolic tangent of input element-wise.	Ascend GPU CPU

mindspore.ops.csr_abs

mindspore.ops.**csr_abs**(*x:* CSRTensor)

Returns csr_absolute value of a CSRTensor element-wise.

$$out_i = |x_i|$$

Parameters

x (CSRTensor) – The input CSRTensor.

Returns

CSRTensor, has the same shape as the x.

Raises

TypeError – If x is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_abs(x)
>>> print(output.values)
[1. 2.]
```

mindspore.ops.csr_acos

mindspore.ops.**csr_acos**(*x:* CSRTensor)

Computes arccosine of input csr_tensors element-wise.

$$out_i = \cos^{-1}(x_i)$$

Parameters

x (CSRTensor) – Input CSRTensor.

Returns

CSRTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64, complex64, complex128.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_acos(x)
>>> print(output.values)
[3.1415927       nan]
```

mindspore.ops.csr_acosh

mindspore.ops.**csr_acosh**(*x:* CSRTensor)

Computes inverse hyperbolic cosine of the inputs element-wise.

$$out_i = \cosh^{-1}(input_i)$$

Parameters

x (CSRTensor) –The input CSRTensor of inverse hyperbolic cosine function, i.e. $input_i$, its element must be in range [1, inf].

Returns

CSRTensor, has the same shape and type as *x*.

Raises

TypeError –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_acosh(x)
>>> print(output.values)
[     nan 1.316958]
```

mindspore.ops.csr_add

mindspore.ops.**csr_add**(*a:* CSRTensor, *b:* CSRTensor, *alpha:* Tensor, *beta:* Tensor)

Computes the linear combination of two input CSRTensors a and b.

$$out = alpha * a + beta * b$$

where both a and b are CSRTensor, $alpha$ and $beta$ are both Tensor

---

Note: The user need to ensure that the input sparse matrix is legal. Otherwise, the behavior of the operator is undefined. For example, when there are multiple elements in the same position, the operator may return an error of fail execute.

Parameters

- **a** (CSRTensor) –Input sparse CSRTensor.
- **b** (CSRTensor) –Input sparse CSRTensor.
- **alpha** (Tensor) –Dense Tensor, its shape must be able to broadcast to a.

- **beta** (Tensor) –Dense Tensor, its shape must be able to broadcast to b.

Returns

CSRTensor, a CSRTensor containing the following parts.

- **indptr** - Indicates the start and end point for non-zero values in each row.
- **indices** - The column positions of all non-zero values of the input.
- **values** - The non-zero values.
- **shape** - The shape of the CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.common.dtype as mstype
>>> from mindspore import Tensor, CSRTensor
>>> from mindspore import ops
>>> a_indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> a_indices = Tensor([0, 1], dtype=mstype.int32)
>>> a_values = Tensor([1, 2], dtype=mstype.float32)
>>> shape = (2, 6)
>>> b_indptr = Tensor([0, 1, 2], dtype=mstype.int32)
>>> b_indices = Tensor([0, 1], dtype=mstype.int32)
>>> b_values = Tensor([1, 2], dtype=mstype.float32)
>>> alpha = Tensor(1, mstype.float32)
>>> beta = Tensor(1, mstype.float32)
>>> csra = CSRTensor(a_indptr, a_indices, a_values, shape)
>>> csrb = CSRTensor(b_indptr, b_indices, b_values, shape)
>>> out = ops.csr_add(csra, csrb, alpha, beta)
>>> print(out)
CSRTensor(shape=[2, 6], dtype=Float32,                    indptr=Tensor(shape=[3],␣
↪dtype=Int32, value=[0 1 2]),                    indices=Tensor(shape=[2], dtype=Int32,␣
↪value=[0 1]),                    values=Tensor(shape=[2], dtype=Float32, value=[ 2.
↪00000000e+00  4.00000000e+00]))
```

mindspore.ops.csr_asin

mindspore.ops.**csr_asin**(*x:* CSRTensor)

Computes arcsine of input csr_tensors element-wise.

$$out_i = \sin^{-1}(x_i)$$

Parameters

x (CSRTensor) –Input CSRTensor. The data types should be one of the following types: float16, float32, float64.

Returns

CSRTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** –If *x* is not a CSRTensor.
- **TypeError** –If dtype of *x* is not float16, float32, float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_asin(x)
>>> print(output.values)
[-1.5707964        nan]
```

mindspore.ops.csr_asinh

mindspore.ops.**csr_asinh**(*x:* CSRTensor)

Computes inverse hyperbolic sine of the input element-wise.

$$out_i = \sinh^{-1}(input_i)$$

Parameters

x (CSRTensor) –The input CSRTensor of inverse hyperbolic sine function, i.e. $input_i$.

Returns

CSRTensor, has the same shape and type as *x*.

Raises

TypeError –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_asinh(x)
>>> print(output.values)
[-0.8813736  1.4436355]
```

mindspore.ops.csr_atan

mindspore.ops.**csr_atan**(*x:* CSRTensor)

Computes the trigonometric inverse tangent of the input element-wise.

$$out_i = \tan^{-1}(x_i)$$

Parameters

x (CSRTensor) – The data type should be one of the following types: float16, float32.

Returns

A CSRTensor, has the same type as the input.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not float16 or float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_atan(x)
>>> print(output.values)
[-0.7853982  1.1071488]
```

mindspore.ops.csr_atanh

mindspore.ops.**csr_atanh**(*x:* CSRTensor)

Computes inverse hyperbolic tangent of the input element-wise.

$$out_i = \tanh^{-1}(x_i)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

x (CSRTensor) – Input CSRTensor. The shape is $(N, *)$ where $*$ means, any number of additional dimensions. The data type should be one of the following types: float16, float32.

Returns

A CSRTensor, has the same type as the input.

Raises

- **TypeError** – If x is not a CSRTensor.
- **TypeError** – If dtype of x is not float16 or float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_atanh(x)
>>> print(output.values)
[-inf  nan]
```

mindspore.ops.csr_ceil

mindspore.ops.**csr_ceil**(*x: CSRTensor*)

Rounds a CSRTensor up to the closest integer element-wise.

$$out_i = \lceil x_i \rceil = \lfloor x_i \rfloor + 1$$

Parameters

x (CSRTensor) – The input CSRTensor with a dtype of float16 or float32.

Returns

CSRTensor, has the same shape as the x.

Raises

- **TypeError** – If x is not a CSRTensor.
- **TypeError** – If dtype of x is not float16 or float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_ceil(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.csr_cos

mindspore.ops.**csr_cos**(*x:* CSRTensor)

Computes cosine of input element-wise.

$$out_i = \cos(x_i)$$

Warning: Currently support data types float16 and float32. If use float64, there may be a problem of missing precision.

Parameters

x (CSRTensor) – Input CSRTensor.

Returns

CSRTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64, complex64, complex128.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_cos(x)
>>> print(output.values)
[ 0.5403023  -0.41614684]
```

mindspore.ops.csr_cosh

mindspore.ops.**csr_cosh**(*x:* CSRTensor)

Computes hyperbolic cosine of input element-wise.

$$out_i = \cosh(x_i)$$

Parameters

x (CSRTensor) –The input CSRTensor of hyperbolic cosine function, its data type must be float16, float32, float64, complex64 or complex128.

Returns

CSRTensor, has the same shape as *x*.

Raises

- **TypeError** –If the dtype of *x* is not one of the following types: float16, float32, float64, complex64, complex128.
- **TypeError** –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_cosh(x)
>>> print(output.values)
[1.5430807 3.7621956]
```

mindspore.ops.csr_exp

mindspore.ops.**csr_exp**(*x:* CSRTensor)

Returns csr_exponential of a CSRTensor element-wise.

$$out_i = e^{x_i}$$

Parameters

x (CSRTensor) –The input CSRTensor.

Returns

CSRTensor, has the same shape and dtype as the *x*.

Raises

TypeError –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_exp(x)
>>> print(output.values)
[0.36787948 7.3890557 ]
```

mindspore.ops.csr_expm1

mindspore.ops.**csr_expm1**(*x: CSRTensor*)

Returns exponential then minus 1 of a CSRTensor element-wise.

$$out_i = e^{x_i} - 1$$

Parameters

x (CSRTensor) – The input CSRTensor with a dtype of float16 or float32.

Returns

CSRTensor, has the same shape as the *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_expm1(x)
>>> print(output.values)
[-0.63212055  6.389056  ]
```

mindspore.ops.csr_floor

mindspore.ops.**csr_floor**(*x:* CSRTensor)

Rounds a CSRTensor down to the closest integer element-wise.

$$out_i = \lfloor x_i \rfloor$$

Parameters

x (CSRTensor) – The input CSRTensor, its data type must be float16, float32 or float64.

Returns

CSRTensor, has the same shape as *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not in [float16, float32, float64].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_floor(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.csr_inv

mindspore.ops.**csr_inv**(*x:* CSRTensor)

Computes Reciprocal of input CSRTensor element-wise.

$$out_i = \frac{1}{x_i}$$

Parameters

x (CSRTensor) – Input CSRTensor. Must be one of the following types: float16, float32 or int32.

Returns

CSRTensor, has the same type and shape as input shape value.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not one of float16, float32, int32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_inv(x)
>>> print(output.values)
[-1.   0.5]
```

mindspore.ops.csr_isfinite

mindspore.ops.**csr_isfinite**(*x:* CSRTensor)

Determines which elements are finite for each position.

$$out_i = \begin{cases} \text{if } x_i = \text{Finite}, & True \\ \text{if } x_i \neq \text{Finite}, & False \end{cases}$$

Parameters

x (CSRTensor) – The input CSRTensor.

Returns

CSRTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError – If x is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_isfinite(x)
>>> print(output.values)
[ True  True]
```

mindspore.ops.csr_isinf

mindspore.ops.**csr_isinf**(*x:* CSRTensor)

Determines which elements are inf or -inf for each position.

$$out_i = \begin{cases} \text{if } x_i = \text{Inf}, & True \\ \text{if } x_i \neq \text{Inf}, & False \end{cases}$$

where Inf means not a number.

Parameters

x (CSRTensor) –The input CSRTensor.

Returns

CSRTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError –If x is not a CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_isinf(x)
>>> print(output.values)
[False False]
```

mindspore.ops.csr_isnan

mindspore.ops.**csr_isnan**(*x:* CSRTensor)

Determines which elements are NaN for each position.

$$out_i = \begin{cases} True, & \text{if } x_i = \text{NaN} \\ False, & \text{if } x_i \neq \text{NaN} \end{cases}$$

where NaN means not a number.

Parameters

x (CSRTensor) –The input CSRTensor.

Returns

CSRTensor, has the same shape of input, and the dtype is bool.

Raises

TypeError –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_isnan(x)
>>> print(output.values)
[False False]
```

mindspore.ops.csr_log

mindspore.ops.**csr_log**(*x:* CSRTensor)

Returns the natural logarithm of a CSRTensor element-wise.

$$y_i = \log_e(x_i)$$

Warning: If the input value of operator Log is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

x (CSRTensor) –Input CSRTensor of any dimension. If the value is smaller than 0, return nan.

Returns

CSRTensor, has the same shape and dtype as the *x*.

Raises

- **TypeError** –If *x* is not a CSRTensor.
- **TypeError** –If dtype of *x* is not float16, float32 or float64 on GPU and CPU.
- **TypeError** –If dtype of *x* is not float16 or float32 on Ascend.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_log(x)
>>> print(output.values)
[       nan 0.69314575]
```

mindspore.ops.csr_log1p

mindspore.ops.**csr_log1p**(*x: CSRTensor*)

Returns the natural logarithm of one plus the input CSRTensor element-wise.

$$out_i = \log\_e(x_i + 1)$$

Parameters

x (CSRTensor) – The input CSRTensor. With float16 or float32 data type. The value must be greater than -1.

Returns

CSRTensor, has the same shape as the *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_log1p(x)
>>> print(output.values)
[     -inf 1.0986123]
```

mindspore.ops.csr_mm

mindspore.ops.**csr_mm**(*a:* CSRTensor, *b:* CSRTensor, *trans_a:* *bool* = *False*, *trans_b:* *bool* = *False*, *adjoint_a:* *bool* = *False*, *adjoint_b:* *bool* = *False*)

Return the matrix multiplication result of the right-multiply matrix (dense or CSRTensor) of the CSRTensor. The CSRTensor with shape *[M, N]* needs to adapt the right matrix with shape *[N, K]* to get the dense matrix or CSRTensor with result *[M, K]*.

Note: If right matrix is CSRTensor, currently only supports GPU backend. If right matrix is Tensor, currently supports CPU backend with LLVM no lower than 12.0.1, and GPU backend.

Parameters

- **a** (CSRTensor) –Sparse CSR Tensor, rank should be 2.
- **b** (CSRTensor) –Sparse CSR Tensor, rank should be 2.
- **trans_a** (*bool, optional*) –whether to transpose CSRTensor a. Default: `False` .
- **trans_b** (*bool, optional*) –whether to transpose CSRTensor b. Default: `False` .
- **adjoint_a** (*bool, optional*) –whether to adjoint CSRTensor a. Default: `False` .
- **adjoint_b** (*bool, optional*) –whether to adjoint CSRTensor b. Default: `False` .

Returns

CSRTensor.

Supported Platforms:

`GPU`

Examples

```
>>> from mindspore import Tensor, CSRTensor
>>> from mindspore import dtype as mstype
>>> from mindspore import ops
>>> a_shape = (4, 5)
>>> a_indptr = Tensor([0, 1, 1, 3, 4], dtype=mstype.int32)
>>> a_indices = Tensor([0, 3, 4, 0],dtype=mstype.int32)
>>> a_values = Tensor([1.0, 5.0, -1.0, -2.0], dtype=mstype.float32)
>>> b_shape = (5, 3)
>>> b_indptr = Tensor([0, 1, 1, 3, 3, 3], dtype=mstype.int32)
>>> b_indices = Tensor([0, 0, 1],dtype=mstype.int32)
>>> b_values = Tensor([2.0, 7.0, 8.0], dtype=mstype.float32)
>>> a = CSRTensor(a_indptr, a_indices, a_values, a_shape)
>>> b = CSRTensor(b_indptr, b_indices, b_values, b_shape)
>>> c = ops.csr_mm(a, b)
>>> print(c.shape)
(4, 3)
>>> print(c.values)
[2. -4.]
>>> print(c.indptr)
[0 1 1 1 2]
>>> print(c.indices)
[0 0]
```

mindspore.ops.csr_neg

mindspore.ops.**csr_neg**(*x:* CSRTensor)

Returns a CSRTensor with csr_negative values of the input CSRTensor element-wise.

$$out_i = -x_i$$

Parameters

x (CSRTensor) – The input CSRTensor with a dtype of Number.

Returns

CSRTensor, has the same shape and dtype as input.

Raises

TypeError – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_neg(x)
>>> print(output.values)
[ 1. -2.]
```

mindspore.ops.csr_relu

mindspore.ops.**csr_relu**(*x:* CSRTensor)

Computes ReLU (Rectified Linear Unit activation function) of input csr_tensors element-wise.

It returns max(x, 0) element-wise. Specially, the neurons with the negative output will be suppressed and the active neurons will stay the same.

$$ReLU(x) = (x)^+ = \max(0, x)$$

Parameters

x (CSRTensor) – Input CSRTensor.

Returns

CSRTensor, with the same dtype and shape as the *x*.

Raises

- **TypeError** – If dtype of *x* is not a number.
- **TypeError** – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_relu(x)
>>> print(output.values)
[0. 2.]
```

mindspore.ops.csr_relu6

mindspore.ops.**csr_relu6**(*x:* CSRTensor)

Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input csr_tensors element-wise.

$$\text{ReLU6}(x) = \min(\max(0, x), 6)$$

It returns $\min(\max(0, x), 6)$ element-wise.

Parameters

x (CSRTensor) – Input CSRTensor, with float16 or float32 data type.

Returns

CSRTensor, with the same dtype and shape as the *x*.

Raises

- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **TypeError** – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_relu6(x)
>>> print(output.values)
[0. 2.]
```

mindspore.ops.csr_round

mindspore.ops.**csr_round**(*x:* CSRTensor)

Returns half to even of a CSRTensor element-wise.

$$out_i \approx x_i$$

Parameters

x (CSRTensor) – The input CSRTensor.

Returns

CSRTensor, has the same shape and type as the *x*.

Raises

TypeError – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_round(x)
>>> print(output.values)
[-1.  2.]
```

mindspore.ops.csr_sigmoid

mindspore.ops.**csr_sigmoid**(*x:* CSRTensor)

Sigmoid activation function.

Computes Sigmoid of input element-wise. The Sigmoid function is defined as:

$$\text{csr\_sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)}$$

where x_i is an element of the x.

Parameters

x (CSRTensor) – Input CSRTensor, the data type is float16, float32, float64, complex64 or complex128.

Returns

CSRTensor, with the same type and shape as the x.

Raises

- **TypeError** – If dtype of *x* is not float16, float32, float64, complex64 or complex128.

- **TypeError** – If *x* is not a CSRTensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_sigmoid(x)
>>> print(output.values)
[0.26894143 0.8807971 ]
```

mindspore.ops.csr_sin

`mindspore.ops.`**`csr_sin`**(*x:* CSRTensor)

Computes sine of the input element-wise.

$$out_i = \sin(x_i)$$

Parameters

x (`CSRTensor`) – Input CSRTensor.

Returns

CSRTensor, has the same shape and dtype as *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64, complex64, complex128.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_sin(x)
>>> print(output.values)
[-0.84147096  0.9092974 ]
```

mindspore.ops.csr_sinh

mindspore.ops.**csr_sinh**(*x:* CSRTensor)

Computes hyperbolic sine of the input element-wise.

$$out_i = \sinh(x_i)$$

Parameters

x (CSRTensor) –The input CSRTensor of hyperbolic sine function.

Returns

CSRTensor, has the same shape as *x*.

Raises

TypeError –If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_sinh(x)
>>> print(output.values)
[-1.1752012  3.6268604]
```

mindspore.ops.csr_softmax

mindspore.ops.**csr_softmax**(*logits:* CSRTensor, *dtype: mstype*)

Calculates the softmax of a CSRTensorMatrix.

Parameters

- **logits** (CSRTensor) –Input sparse CSRTensor.
- **dtype** (dtype) –Input data type.

Returns

CSRTensor, a CSRTensor containing

- **indptr** - Indicates the start and end point for non-zero values in each row.
- **indices** - The column positions of all non-zero values of the input.
- **values** - The non-zero values of the dense tensor.
- **shape** - The shape of the CSRTensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> import mindspore.common.dtype as mstype
>>> from mindspore import Tensor, CSRTensor
>>> logits_indptr = Tensor([0, 4, 6], dtype=mstype.int32)
>>> logits_indices = Tensor([0, 2, 3, 4, 3, 4], dtype=mstype.int32)
>>> logits_values = Tensor([1, 2, 3, 4, 1, 2], dtype=mstype.float32)
>>> shape = (2, 6)
>>> logits = CSRTensor(logits_indptr, logits_indices, logits_values, shape)
>>> out = ops.csr_softmax(logits, dtype=mstype.float32)
>>> print(out)
CSRTensor(shape=[2, 6], dtype=Float32, indptr=Tensor(shape=[3], dtype=Int32, value=[0 4 6]),
              indices=Tensor(shape=[6], dtype=Int32, value=[0 2 3 4 3 4]),
              values=Tensor(shape=[6], dtype=Float32, value=[ 3.20586003e-02  8.71443152e-
↪02  2.36882806e-01
              6.43914223e-01  2.68941432e-01  7.31058598e-01]))
```

mindspore.ops.csr_softsign

mindspore.ops.**csr_softsign**(*x:* CSRTensor)

Softsign activation function.

The function is shown as follows:

$$\text{SoftSign}(x) = \frac{x}{1 + |x|}$$

Parameters

x (CSRTensor) – Input CSRTensor, with float16 or float32 data type.

Returns

CSRTensor, with the same type and shape as the *x*.

Raises

- **TypeError** – If *x* is not a CSRTensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_softsign(x)
>>> print(output.values)
[-0.5        0.6666667]
```

mindspore.ops.csr_sqrt

mindspore.ops.**csr_sqrt**(*x:* CSRTensor)

Returns sqrt of a CSRTensor element-wise.

$$out_i = \sqrt{x_i}$$

Parameters

x (CSRTensor) – The input CSRTensor with a dtype of Number.

Returns

CSRTensor, has the same shape and dtype as the *x*.

Raises

TypeError – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_sqrt(x)
>>> print(output.values)
[      nan 1.4142135]
```

mindspore.ops.csr_square

mindspore.ops.**csr_square**(*x:* CSRTensor)

Returns square of a CSRTensor element-wise.

$$out_i = (x_i)^2$$

Parameters

x (CSRTensor) – The input CSRTensor with a dtype of Number.

Returns

CSRTensor, has the same shape and dtype as the *x*.

Raises

TypeError – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_square(x)
>>> print(output.values)
[1. 4.]
```

mindspore.ops.csr_tan

mindspore.ops.**csr_tan**(*x:* CSRTensor)

Computes tangent of *x* element-wise.

$$out_i = \tan(x_i)$$

Parameters

x (CSRTensor) – The input CSRTensor.

Returns

CSRTensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_tan(x)
>>> print(output.values)
[-1.5574077 -2.1850398]
```

mindspore.ops.csr_tanh

mindspore.ops.**csr_tanh**(*x:* CSRTensor)

Computes hyperbolic tangent of input element-wise. The Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input CSRTensor.

Parameters

x (CSRTensor) – Input CSRTensor, with float16 or float32 data type.

Returns

CSRTensor, with the same type and shape as the *x*.

Raises

- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **TypeError** – If *x* is not a CSRTensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, CSRTensor
>>> indptr = Tensor([0, 1, 2, 2], dtype=mstype.int32)
>>> indices = Tensor([3, 0], dtype=mstype.int32)
>>> values = Tensor([-1, 2], dtype=mstype.float32)
>>> shape = (3, 4)
>>> x = CSRTensor(indptr, indices, values, shape)
>>> output = ops.csr_tanh(x)
>>> print(output.values)
[-0.7615942  0.9640276]
```

2.8 MC2 Functions

Warning: These are experimental APIs that are subject to change or deletion.

API Name	Description	Supported Platforms
mindspore.ops.all_gather_matmul	In the TP segmentation scenario, allgather and matmul are fused, and communication and computational pipelines are parallelized within the fusion operator.	Ascend
mindspore.ops.matmul_reduce_scatter	In the TP segmentation scenario, matmul and reducescatter are fused, and communication and computational pipelines are parallelized within the fusion operator.	Ascend

2.8.1 mindspore.ops.all_gather_matmul

mindspore.ops.**all_gather_matmul**(*input*, *x2*, *group*, *world_size*, ***, *bias=None*, *gather_index=0*, *gather_output=True*, *comm_turn=0*, *trans_input=False*, *trans_x2=False*) → *Tensor*

In the TP segmentation scenario, allgather and matmul are fused, and communication and computational pipelines are parallelized within the fusion operator.

$$output = allgather(input)@x2$$
$$gather\_out = allgather(input)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −The left matrix of matmul, the dtype supports float16 and bfloat16, the shape supports 2 dimensions, and the data format supports ND.
- **x2** (Tensor) −The right matrix of matmul, the dtype needs to be consistent with `input` , the shape supports 2 dimensions, and the data format supports ND.
- **group** (*str*) −Communication group name, can be created by `create_group` method, or use the default group `mindspore.communication.GlobalComm.WORLD_COMM_GROUP`.
- **world_size** (*int*) −The total number of ranks in the communication group, should be consistent with the number of devices actually running, supporting `2` , `4` , and `8` .

Keyword Arguments

- **bias** (Tensor, *optional*) −Currently only `None` is supported. Default: `None` .
- **gather_index** (int, *optional*) −Indicates the allgather operation object, `0` means gather `input` , `1` means gather `x2` . Currently only `0` is supported. Default: `0` .
- **gather_output** (bool, *optional*) −Indicates whether gather output is required. Default: `True` .
- **comm_turn** (int, *optional*) −Indicates the granularity of communication between ranks. Currently only `0` is supported. Default: `0` .

- **trans_input** (bool, *optional*) –Indicates whether input is transposed. Currently only False is supported. Default: False .
- **trans_x2** (bool, *optional*) –Indicates whether x2 is transposed. Default: False .

Returns

- output (Tensor) - The result of allgather and matmul fusion calculations.
- gather_out (Tensor) - The result of allgather. If gather_output is False , gather_out returns a tensor with shape 0.

Note:

- When using this interface, please ensure that the driver firmware package and CANN package are both the matching 8.0.RC2 version or a higher version, otherwise an error will be reported, such as BUS ERROR.
- The shape of input is (m, k), the shape of x2 is (k, n), k is required to be equal, and the value range of k is [256, 65535). The shape of output is (m * world_size, n), and the shape of gather_out is (m * world_size, k).
- The common fusion operators in a model only support the same communication group.

Raises

- **TypeError** –Any arg is of wrong type.
- **RuntimeError** –The dtype of input or x2 is neither float16 nor bfloat16.
- **RuntimeError** –The dtypes of input and x2 are different.
- **RuntimeError** –The shape of input or x2 is not two-dimensional.
- **RuntimeError** –The k axis of input shape and x2 shape are not equal.
- **RuntimeError** –k is less than 256 or greater than or equal to 65535 .
- **RuntimeError** –bias is not None.
- **RuntimeError** –group does not exist.
- **RuntimeError** –world_size is inconsistent with the actual number of running cards.
- **RuntimeError** –world_size is not equal to 2 , 4 , or 8 .
- **RuntimeError** –gather_index is not 0 .
- **RuntimeError** –trans_input is True .

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import ops
>>> ms.communication.init()
>>> rank = ms.communication.get_rank()
>>> np.random.seed(rank)
>>> input = ms.Tensor(np.random.randn(128, 256).astype(np.float32), dtype=ms.float16)
>>> x2 = ms.Tensor(np.random.randn(256, 512).astype(np.float32), dtype=ms.float16)
>>> group = ms.communication.GlobalComm.WORLD_COMM_GROUP
>>> world_size = ms.communication.get_group_size()
>>> output, gather_out = ops.all_gather_matmul(
...     input,
...     x2,
...     group,
...     world_size,
...     bias=None,
...     gather_index=0,
...     gather_output=True,
...     comm_turn=0,
...     trans_input=False,
...     trans_x2=False,
... )
>>> print(output.shape)
(256, 512)
>>> print(gather_out.shape)
(256, 256)
```

2.8.2 mindspore.ops.matmul_reduce_scatter

mindspore.ops.**matmul_reduce_scatter**(*input*, *x2*, *group*, *world_size*, *, *reduce_op='sum'*, *bias=None*, *comm_turn=0*, *trans_input=False*, *trans_x2=False*) → *Tensor*

In the TP segmentation scenario, matmul and reducescatter are fused, and communication and computational pipelines are parallelized within the fusion operator.

$$output = reducescatter(input@x2)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – The left matrix of matmul, the dtype supports float16 and bfloat16, the shape supports 2 dimensions, and the data format supports ND.

- **x2** (Tensor) –The right matrix of matmul, the dtype needs to be consistent with `input` , the shape supports 2 dimensions, and the data format supports ND.
- **group** (*str*) –Communication group name, can be created by `create_group` method, or use the default group `mindspore.communication.GlobalComm.WORLD_COMM_GROUP`.
- **world_size** (*int*) –The total number of ranks in the communication group, should be consistent with the number of devices actually running, supporting `2` , `4` , and `8` .

Keyword Arguments

- **reduce_op** (*str, optional*) – `'sum'` .
- **bias** (Tensor, *optional*) –Currently only `None` is supported. Default: `None` .
- **comm_turn** (int, *optional*) –Indicates the granularity of communication between ranks. Currently only `0` is supported. Default: `0` .
- **trans_input** (bool, *optional*) –Indicates whether `input` is transposed. Currently only `False` is supported. Default: `False` .
- **trans_x2** (bool, *optional*) –Indicates whether `x2` is transposed. Default: `False` .

Returns

- output (Tensor) - The result of allgather and matmul fusion calculations.

Note:

- When using this interface, please ensure that the driver firmware package and CANN package are both the matching 8.0.RC2 version or a higher version, otherwise an error will be reported, such as BUS ERROR.
- The shape of `input` is (m, k), the shape of `x2` is (k, n), k is required to be equal, and the value range of k is [256, 65535), and m is required to be an integer multiple of `world_size` . The shape of `output` is (m * world_size, n).
- The common fusion operators in a model only support the same communication group.

Raises

- **TypeError** –Any arg is of wrong type.
- **RuntimeError** –The dtype of `input` or `x2` is neither float16 nor bfloat16.
- **RuntimeError** –The dtypes of `input` and `x2` are different.
- **RuntimeError** –The shape of `input` or `x2` is not two-dimensional.
- **RuntimeError** –The k axis of `input` shape and `x2` shape are not equal.
- **RuntimeError** –k is less than `256` or greater than or equal to `65535` .
- **RuntimeError** –`bias` is not None.
- **RuntimeError** –`group` does not exist.
- **RuntimeError** –`world_size` is inconsistent with the actual number of running cards.
- **RuntimeError** –`world_size` is not equal to `2` , `4` , or `8` .
- **RuntimeError** –`reduce_op` is not `'sum'` .
- **RuntimeError** –`trans_input` is `True` .

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> import numpy as np
>>> ms.communication.init()
>>> rank = ms.communication.get_rank()
>>> np.random.seed(rank)
>>> input = ms.Tensor(np.random.randn(1024, 256).astype(np.float32), dtype=ms.float16)
>>> x2 = ms.Tensor(np.random.randn(256, 512).astype(np.float32), dtype=ms.float16)
>>> group = ms.communication.GlobalComm.WORLD_COMM_GROUP
>>> world_size = ms.communication.get_group_size()
>>> reduce_op = ops.ReduceOp.SUM
>>> output = ops.matmul_reduce_scatter(
...     input,
...     x2,
...     group,
...     world_size,
...     reduce_op=reduce_op,
...     bias=None,
...     comm_turn=0,
...     trans_input=False,
...     trans_x2=False,
... )
>>> print(output.shape)
(512, 512)
```

CHAPTER

THREE

MINDSPORE.NN

Neural Network Cell

For building predefined building blocks or computational units in neural networks.

For more information about dynamic shape support status, please refer to Dynamic Shape Support Status of nn Interface .

Compared with the previous version, the added, deleted and supported platforms change information of *mindspore.nn* operators in MindSpore, please refer to the link mindspore.nn API Interface Change .

3.1 Basic Block

API Name	Description	Supported Platforms
mindspore.nn.Cell	The basic building block of neural networks in MindSpore.	Ascend GPU CPU
mindspore.nn.GraphCell	Base class for running the graph loaded from a MindIR.	Ascend GPU CPU
mindspore.nn.LossBase	Base class for other losses.	Ascend GPU CPU
mindspore.nn.Optimizer	Base class for updating parameters.	Ascend GPU CPU

3.1.1 mindspore.nn.Cell

class mindspore.nn.**Cell**(*auto_prefix=True*, *flags=None*)

The basic building block of neural networks in MindSpore. The model or neural network layer should inherit this base class.

Layers in *mindspore.nn* are also the subclass of Cell, such as *mindspore.nn.Conv2d*, and *mindspore.nn.ReLU*, etc. Cell will be compiled into a calculation graph in GRAPH_MODE (static graph mode) and used as the basic module of neural networks in PYNATIVE_MODE (dynamic graph mode).

Note: Cell is the inference mode by default. For a class that inherits a Cell, if the training and inference have different structures, the subclass performs the inference branch by default. To set the training mode, refer to *mindspore.nn.Cell.set_train()* .

Warning: In the subclass of Cell, it's not allowed to define a method named 'cast' and not allowed to define an attribute named 'phase' or 'cells', otherwise, an error will be raised.

Parameters

- **auto_prefix** (*bool, optional*) –Whether to automatically generate NameSpace for Cell and its child cells. It also affects the names of parameters in the *Cell*. If set to `True` , the parameter name will be automatically prefixed, otherwise not. In general, the backbone network should be set to `True` , otherwise the duplicate name problem will appear. The cell to train the backbone network, such as optimizer and *mindspore.nn.TrainOneStepCell*, should be set to `False` , otherwise the parameter name in backbone will be changed by mistake. Default: `True` .
- **flags** (*dict, optional*) –Network configuration information, currently it is used for the binding of network and dataset. Users can also customize network attributes by this parameter. Default: `None` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> class MyCell(nn.Cell):
...     def __init__(self, forward_net):
...         super(MyCell, self).__init__(auto_prefix=False)
...         self.net = forward_net
...         self.relu = ops.ReLU()
...
...     def construct(self, x):
...         y = self.net(x)
...         return self.relu(y)
>>>
>>> inner_net = nn.Conv2d(120, 240, 4, has_bias=False, weight_init='normal')
>>> my_net = MyCell(inner_net)
>>> print(my_net.trainable_params())
... # If the 'auto_prefix' set to True or not set when call the '__init__' method of the␣
↪parent class,
... # the parameter's name will be 'net.weight'.
[Parameter (name=weight, shape=(240, 120, 4, 4), dtype=Float32, requires_grad=True)]
```

add_flags (***flags*)

Add customized attributes for cell.

This method is also called when the cell class is instantiated and the class parameter 'flags' is set to True.

Parameters

flags (*dict*) –Network configuration information, currently it is used for the binding of network and dataset. Users can also customize network attributes by this parameter.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...
...     def construct(self, x):
```

(continues on next page)

(continued from previous page)

```
...         x = self.relu(x)
...         return x
>>> net = Net()
>>> net.add_flags(sink_mode=True)
>>> print(net.sink_mode)
True
```

add_flags_recursive(***flags*)

If a cell contains child cells, this method can recursively customize attributes of all cells.

Parameters

flags (*dict*) – Network configuration information, currently it is used for the binding of network and dataset. Users can also customize network attributes by this parameter.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...
...     def construct(self, x):
...         x = self.relu(x)
...         return x
>>> net = Net()
>>> net.add_flags_recursive(sink_mode=True)
>>> print(net.sink_mode)
True
```

apply(*fn*)

Applies fn recursively to every subcell (as returned by .cells()) as well as self. Typical use includes initializing the parameters of a model.

Parameters

fn (*function*) – function to be applied to each subcell.

Returns

Cell, self.

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore.common.initializer import initializer, One
>>> net = nn.SequentialCell(nn.Dense(2, 2), nn.Dense(2, 2))
>>> def func(cell):
...     if isinstance(cell, nn.Dense):
...         cell.weight.set_data(initializer(One(), cell.weight.shape, cell.weight.
↪dtype))
>>> net.apply(func)
```

(continues on next page)

(continued from previous page)

```
SequentialCell(
  (0): Dense(input_channels=2, output_channels=2, has_bias=True)
  (1): Dense(input_channels=2, output_channels=2, has_bias=True)
)
>>> print(net[0].weight.asnumpy())
[[1. 1.]
 [1. 1.]]
```

auto_cast_inputs(*inputs*)

Auto cast inputs in mixed precision scenarios.

Parameters

inputs (*tuple*) – the inputs of construct.

Returns

Tuple, the inputs after data type cast.

property bprop_debug

Get whether cell custom bprop debug is enabled.

buffers(*recurse: bool = True*)

Return an iterator over cell buffers.

Parameters

recurse (*bool, optional*) – If `True` , then yields buffers of this cell and all sub cells. Otherwise, yields only buffers that are direct members of this cell. Default `True`.

Returns

Iterator[Tensor], an iterator of buffer.

Examples

```
>>> import mindspore
...
...
>>> class NetB(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_b", mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.buffer_b
...
...
>>> class NetA(mindspore.nn.Cell):
...     def __init__(self, net_b):
...         super().__init__()
...         self.net_b = net_b
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...
...     def construct(self, x):
...         return self.net_b(x) + self.buffer_a
...
...
```

(continues on next page)

(continued from previous page)

```
>>> net_b = NetB()
>>> net_a = NetA(net_b)
>>>
>>> for buffer in net_a.buffers():
...     print(f'buffer is {buffer}')
buffer is [4 5 6]
buffer is [1 2 3]
```

cast_inputs(*inputs*, *dst_type*)

Cast inputs to specified type.

Parameters

- **inputs** (*tuple[Tensor]*) –The cell inputs.
- **dst_type** (mindspore.dtype) –The specified data type.

Returns

tuple[Tensor], the result with destination data type.

cast_param(*param*)

Cast parameter according to auto mix precision level in pynative mode.

This interface is currently used in the case of auto mix precision and usually needs not to be used explicitly.

Parameters

param (Parameter) –Parameters, the type of which should be cast.

Returns

Parameter, the input parameter with type automatically cast.

cells()

Returns an iterator over immediate cells.

Returns

Iteration, the immediate cells in the cell.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dense = nn.Dense(2, 2)
...
...     def construct(self, x):
...         x = self.dense(x)
...         return x
>>> net = Net()
>>> print(net.cells())
odict_values([Dense(input_channels=2, output_channels=2, has_bias=True)])
```

cells_and_names(*cells=None*, *name_prefix=''*)

Returns an iterator over all cells in the network, including the cell's name and itself.

Parameters

- **cells** (*str*) –Cells to iterate over. Default: None .
- **name_prefix** (*str*) –Namespace. Default: '' .

Returns

Iteration, all the child cells and corresponding names in the cell.

Examples

```
>>> from mindspore import nn
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.conv = nn.Conv2d(3, 64, 3)
...     def construct(self, x):
...         out = self.conv(x)
...         return out
>>> names = []
>>> n = Net()
>>> for m in n.cells_and_names():
...     if m[0]:
...         names.append(m[0])
```

check_names()
Check the names of cell parameters.

compile(**args*, ***kwargs*)
Compile Cell as a computation graph, the input must be consistent with the input defined in construct.

Parameters

- **args** (*tuple*) –Args of the Cell object.
- **kwargs** (*dict*) –Kwargs of the Cell object.

compile_and_run(**args*, ***kwargs*)
Compile and run Cell, the input must be consistent with the input defined in construct.

Note: It is not recommended to call directly.

Parameters

- **args** (*tuple*) –Args of the Cell object.
- **kwargs** (*dict*) –Kwargs of the Cell object.

Returns

Object, the result of executing.

construct(**args*, ***kwargs*)
Defines the computation to be performed. This method must be overridden by all subclasses.

Note: It is not supported currently that inputs contain both tuple and non-tuple types at same time.

Parameters

- **args** (*tuple*) –Tuple of variable parameters.
- **kwargs** (*dict*) –Dictionary of variable keyword parameters.

Returns

Tensor, returns the computed result.

extend_repr()
Expand the description of Cell.

To print customized extended information, re-implement this method in your own cells.

flatten_weights(*fusion_size=0*)
Reset data for weight parameters so that they are using contiguous memory chunks grouped by data type.

Note: By default, parameters with same data type will using a single contiguous memory chunk. but for some models with huge number of parameters, splitting a large memory chunk into several smaller memory chunks has the potential for performance gains, if this is the case, we can use 'fusion_size' to limit the maximum memory chunk size.

Parameters

fusion_size (*int*) –Maximum memory chunk size in bytes, 0 for unlimited. Default: 0 .

generate_scope()
Generate the scope for each cell object in the network.

get_buffer(*target: str*)
Return the buffer given by *target* if it exists, otherwise throw an error.

See the docstring for *get_sub_cell* for a more detailed explanation of this method's functionality as well as how to correctly specify *target* .

Parameters

target (*str*) –The fully-qualified string name of the buffer to look for. (See *get_sub_cell* for how to specify a fully-qualified string.)

Returns

Tensor

Examples

```
>>> import mindspore
...
...
>>> class NetC(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_c", mindspore.tensor([0, 0, 0]))
...
...     def construct(self, x):
...         return x + self.buffer_c
...
...
```

(continues on next page)

(continued from previous page)

```
>>> class NetB(mindspore.nn.Cell):
...     def __init__(self, net_c):
...         super().__init__()
...         self.net_c = net_c
...         self.register_buffer("buffer_b", mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return self.net_c(x) + self.buffer_b
...
...
>>> class NetA(mindspore.nn.Cell):
...     def __init__(self, net_b):
...         super().__init__()
...         self.net_b = net_b
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...
...     def construct(self, x):
...         return self.net_b(x) + self.buffer_a
...
...
>>> net_c = NetC()
>>> net_b = NetB(net_c)
>>> net_a = NetA(net_b)
>>> buffer_c = net_a.get_buffer("net_b.net_c.buffer_c")
>>> print(f'buffer_c is {buffer_c}')
buffer_c is [0 0 0]
```

get_extra_state()

Return any extra state to include in the cell's state_dict.

This function is called from `state_dict`. Implement this and a corresponding `set_extra_state` for your cell if you need to store extra state.

Note that extra state should be picklable to ensure working serialization of the state_dict. Only provide backwards compatibility guarantees for serializing tensors; other objects may break backwards compatibility if their serialized pickled form changes.

Returns

object, any extra state to store in the cell's state_dict.

get_flags()

Get the self_defined attributes of the cell, which can be added by *add_flags* method.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...
...     def construct(self, x):
...         x = self.relu(x)
```

(continues on next page)

(continued from previous page)

```
...         return x
>>> net = Net()
>>> net.add_flags(sink_mode=True)
>>> print(net.get_flags())
{'sink_mode':True}
```

get_func_graph_proto()
Return graph binary proto.

get_inputs()
Returns the dynamic_inputs of a cell object in one network.

Returns

inputs (tuple), Inputs of the Cell object.

> **Warning:** This is an experimental API that is subject to change or deletion.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor
>>>
>>> class ReluNet(nn.Cell):
...     def __init__(self):
...         super(ReluNet, self).__init__()
...         self.relu = nn.ReLU()
...     def construct(self, x):
...         return self.relu(x)
>>>
>>> net = ReluNet()
>>> input_dyn = Tensor(shape=[3, None], dtype=ms.float32)
>>> net.set_inputs(input_dyn)
>>> get_inputs = net.get_inputs()
>>> print(get_inputs)
(Tensor(shape=[3, -1], dtype=Float32, value= ),)
```

get_parameters(*expand=True*)
Returns an iterator over cell parameters.

Yields parameters of this cell. If *expand* is `true` , yield parameters of this cell and all subcells. For more details about subcells, please see the example below.

Parameters

expand (*bool*) – If `true` , yields parameters of this cell and all subcells. Otherwise, only yield parameters that are direct members of this cell. Default: `True` .

Returns

Iteration, all parameters at the cell.

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn, ops, Tensor
>>> import numpy as np
>>> class TestNet(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.my_w1 = ms.Parameter(Tensor(np.ones([4, 4]), ms.float32))
...         self.my_w2 = ms.Parameter(Tensor(np.ones([16]), ms.float32))
...     def construct(self, x):
...         x += self.my_w1
...         x = ops.reshape(x, (16,)) - self.my_w2
...         return x
>>> class TestNet2(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.my_t1 = ms.Parameter(Tensor(np.ones([4, 4]), ms.float32))
...         # self.subcell is a subcell of TestNet2, when using expand=True, the␣
↪parameters of TestNet will
...         # also be gathered.
...         self.subcell = TestNet()
...     def construct(self, x):
...         x += self.my_w1
...         x = ops.reshape(x, (16,)) - self.my_w2
...         return x
>>> net = TestNet2()
>>> print([p for p in net.get_parameters(expand=True)])
[Parameter (name=my_t1, shape=(4, 4), dtype=Float32, requires_grad=True), Parameter␣
↪(name=subcell.my_w1,
shape=(4, 4), dtype=Float32, requires_grad=True), Parameter (name=subcell.my_w2,␣
↪shape=(16,), dtype=Float32,
requires_grad=True)]
```

get_scope()

Returns the scope of a cell object in one network.

Returns

String, scope of the cell.

get_sub_cell(*target: str*)

Return the sub cell given by *target* if it exists, otherwise throw an error.

For example, let's say you have an `nn.Cell` `A` that looks like this:

```
A(
    (net_b): NetB(
        (net_c): NetC(
            (conv): Conv2d(16, 33, kernel_size=(3, 3), stride=(2, 2))
        )
        (dense): Dense(in_features=100, out_features=200, bias=True)
    )
)
```

(The diagram shows an `nn.Cell` `A`. `A` has a nested sub cell `net_b`, which itself has two sub cells `net_c` and `dense`. `net_c` then has a sub cell `conv`.)

To check whether we have the `dense` sub cell, we would call *get_sub_cell("net_b.dense")*. To check whether we have the `conv` sub cell, we would call *get_sub_cell("net_b.net_c.conv")*.

The runtime of `get_sub_cell` is bounded by the degree of cell nesting in *target*. A query against *name_cells* achieves the same result, but it is O(N) in the number of transitive cells. So, for a simple check to see if some sub cells exist, `get_sub_cell` should always be used.

Parameters

target (*str*) –The fully-qualified string name of the sub cell to look for. (See above example for how to specify a fully-qualified string.)

Returns

Cell

Examples

```
>>> import mindspore
...
...
>>> class NetC(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_c", mindspore.tensor([0, 0, 0]))
...         self.dense_c = mindspore.nn.Dense(5, 3)
...
...     def construct(self, x):
...         return self.dense_c(x) + self.buffer_c
...
...
>>> class NetB(mindspore.nn.Cell):
...     def __init__(self, net_c):
...         super().__init__()
...         self.net_c = net_c
...         self.register_buffer("buffer_b", mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return self.net_c(x) + self.buffer_b
...
...
>>> class NetA(mindspore.nn.Cell):
...     def __init__(self, net_b):
...         super().__init__()
...         self.net_b = net_b
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...
...     def construct(self, x):
...         return self.net_b(x) + self.buffer_a
...
...
>>> net_c = NetC()
>>> net_b = NetB(net_c)
>>> net_a = NetA(net_b)
>>> net_c = net_a.get_sub_cell("net_b.net_c")
>>> print(f'net_c is {net_c}')
net_c is NetC(
    (dense_c): Dense(input_channels=5, output_channels=3, has_bias=True)
)
```

infer_param_pipeline_stage()
Infer pipeline stages of all parameters in the cell.

Note:

- The interface is deprecated from version 2.3 and will be removed in a future version.

Returns

The params belong to current stage in pipeline parallel.

Raises

RuntimeError –If there is a parameter does not belong to any stage.

init_parameters_data(*auto_parallel_mode=False*)
Initialize all parameters and replace the original saved parameters in cell.

Note: trainable_params() and other similar interfaces may return different parameter instance after *init_parameters_data*. It is not recommended to save these results.

Parameters

auto_parallel_mode (*bool*) –If running in auto_parallel_mode. Default: `False` .

Returns

Dict[Parameter, Parameter], returns a dict of original parameter and replaced parameter.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dense = nn.Dense(2, 2)
...
...     def construct(self, x):
...         x = self.dense(x)
...         return x
>>> net = Net()
>>> print(net.init_parameters_data())
{Parameter (name=dense.weight, shape=(2,2), dtype=Float32, requires_grad=True):
 Parameter (name=dense.weight, shape=(2,2), dtype=Float32, requires_grad=True),
 Parameter (name=dense.bias, shape=(2,), dtype=Float32, requires_grad=True):
 Parameter (name=dense.bias, shape=(2,), dtype=Float32, requires_grad=True)}
```

insert_child_to_cell(*child_name*, *child_cell*)
Adds a child cell to the current cell with a given name.

Parameters

- **child_name** (*str*) –Name of the child cell.

- **child_cell** (Cell) –The child cell to be inserted.

Raises

- **KeyError** –Child Cell's name is incorrect or duplicated with the other child name.
- **TypeError** –If type of *child_name* is not str.
- **TypeError** –Child Cell's type is incorrect.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> net1 = nn.ReLU()
>>> net2 = nn.Dense(2, 2)
>>> net1.insert_child_to_cell("child", net2)
>>> print(net1)
ReLU(
  (child): Dense(input_channels=2, output_channels=2, has_bias=True)
)
```

insert_param_to_cell(*param_name*, *param*, *check_name_contain_dot=True*)

Adds a parameter to the current cell.

Inserts a parameter with given name to the cell. The method is currently used in *mindspore.nn.Cell.__setattr__*.

Parameters

- **param_name** (*str*) –Name of the parameter.
- **param** (Parameter) –Parameter to be inserted to the cell.
- **check_name_contain_dot** (*bool*) –Determines whether the name input is compatible. Default: True .

Raises

- **KeyError** –If the name of parameter is null or contains dot.
- **TypeError** –If the type of parameter is not Parameter.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, Parameter
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...
...     def construct(self, x):
...         x = self.relu(x)
...         return x
>>> net = Net()
>>> net.insert_param_to_cell("bias", Parameter(Tensor([1, 2, 3])))
>>> print(net.bias)
Parameter(name=bias, shape=(3,), dtype=Int64, requires_grad=True)
```

load_state_dict(*state_dict: Mapping[str, Any]*, *strict: bool = True*)

Copy parameters and buffers from *state_dict* into this cell and its descendants.

If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this cell's *mindspore.nn.Cell.state_dict()* function.

Parameters

- **state_dict** (*dict*) –A dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) –Whether to strictly enforce that the keys in input *state_dict* match the keys returned by this cell's *mindspore.nn.Cell.state_dict()* function. Default `True` .

Returns

A namedtuple with `missing_keys` and `unexpected_keys` fields,

- *missing_keys* is a list of str containing any keys that are expected by this cell but missing from the provided `state_dict`.
- *unexpected_keys* is a list of str containing the keys that are not expected by this cell but present in the provided `state_dict`.

Note: If *strict* is `True` and a parameter or buffer is registered as `None`, but its corresponding key exists in *state_dict*, and *mindspore.nn.Cell.load_state_dict()* will raise a `RuntimeError`.

Examples

```
>>> import mindspore
>>> import os
>>> class Model(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...         self.param_a = mindspore.Parameter(mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.buffer_a + self.param_a
...
...
>>> model = Model()
>>> print(model.state_dict())
>>> mindspore.save_checkpoint(model.state_dict(), './model_state_dict_ckpt')
>>> new_model = Model()
>>> new_model.load_state_dict(mindspore.load_checkpoint('./model_state_dict_ckpt'))
>>> print(new_model.state_dict())
>>> os.remove('./model_state_dict_ckpt')
OrderedDict([('param_a', Parameter (name=param_a, shape=(3,), dtype=Int64, requires_
↪grad=True)), \
('buffer_a', Tensor(shape=[3], dtype=Int64, value= [4, 5, 6]))])
OrderedDict([('param_a', Parameter (name=param_a, shape=(3,), dtype=Int64, requires_
↪grad=True)), \
('buffer_a', Tensor(shape=[3], dtype=Int64, value= [4, 5, 6]))])
```

name_cells()

Returns an iterator over all immediate cells in the network.

Include name of the cell and cell itself.

Returns

Dict, all the child cells and corresponding names in the cell.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dense = nn.Dense(2, 2)
...
...     def construct(self, x):
...         x = self.dense(x)
...         return x
>>> net = Net()
>>> print(net.name_cells())
OrderedDict([('dense', Dense(input_channels=2, output_channels=2, has_bias=True))])
```

named_buffers(*prefix: str = ''*, *recurse: bool = True*, *remove_duplicate: bool = True*)

Return an iterator over cell buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str, optional*) – prefix to prepend to all buffer names. Default `""`.
- **recurse** (*bool, optional*) – if `True` , then yields buffers of this cell and all sub cells. Otherwise, yields only buffers that are direct members of this cell. Default `True`.
- **remove_duplicate** (*bool, optional*) – Whether to remove the duplicated buffers in the result. Default `True`.

Returns

Iterator[Tuple[str, Tensor]], an iterator of tuple containing the name and buffer.

Examples

```
>>> import mindspore
...
...
>>> class NetB(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_b", mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.buffer_b
...
...
>>> class NetA(mindspore.nn.Cell):
...     def __init__(self, net_b):
...         super().__init__()
```

(continues on next page)

(continued from previous page)

```
...         self.net_b = net_b
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...
...     def construct(self, x):
...         return self.net_b(x) + self.buffer_a
...
...
>>> net_b = NetB()
>>> net_a = NetA(net_b)
>>>
>>> for name, buffer in net_a.named_buffers():
...     print(f'buffer name is {name}, buffer is {buffer}')
buffer name is buffer_a, buffer is [4 5 6]
buffer name is net_b.buffer_b, buffer is [1 2 3]
```

offload(*backward_prefetch='Auto'*)

Set the cell offload. All primitive ops in the cell will be set offload. For the intermediate activations calculated by these primitive ops, we will not save them in the forward pass, but offload them and onload them in the backward pass.

Note:

- If Cell.offload is called, the mode should be set to "GRAPH_MODE".
- If Cell.offload is called, lazyinline should be enabled.

Parameters

backward_prefetch (*Union[str, int], optional*) – The timing for prefetching activations in advance in backward pass. Default: `"Auto"`. If set it to `"Auto"`, framework will start to prefetch activations one operator in advance. If set it to a positive int value, framework will start to prefetch activations `backward_prefetch` operators in advance, such as 1, 20, 100.

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore.common import Tensor, Parameter
>>> from mindspore.common.lazy_inline import lazy_inline
>>>
>>> class Block(nn.Cell):
...     def __init__(self):
...         super(Block, self).__init__()
...         self.transpose1 = ops.Transpose()
...         self.transpose2 = ops.Transpose()
...         self.transpose3 = ops.Transpose()
...         self.transpose4 = ops.Transpose()
...         self.real_div1 = ops.RealDiv()
...         self.real_div2 = ops.RealDiv()
...         self.batch_matmul1 = ops.BatchMatMul()
...         self.batch_matmul2 = ops.BatchMatMul()
...         self.softmax = ops.Softmax(-1)
...         self.expand_dims = ops.ExpandDims()
...         self.sub = ops.Sub()
...         self.y = Parameter(Tensor(np.ones((1024, 128, 128)).astype(np.float32)))
```

(continues on next page)

(continued from previous page)

```
...     def construct(self, x):
...         transpose1 = self.transpose1(x, (0, 2, 1, 3))
...         real_div1 = self.real_div1(transpose1, Tensor(2.37891))
...         transpose2 = self.transpose2(x, (0, 2, 3, 1))
...         real_div2 = self.real_div2(transpose2, Tensor(2.37891))
...         batch_matmul1 = self.batch_matmul1(real_div1, real_div2)
...         expand_dims = self.expand_dims(self.y, 1)
...         sub = self.sub(Tensor([1.0]), expand_dims)
...         soft_max = self.softmax(sub)
...         transpose3 = self.transpose3(x, (0, 2, 1, 3))
...         batch_matmul2 = self.batch_matmul2(soft_max[0], transpose3)
...         transpose4 = self.transpose4(batch_matmul2, (0, 2, 1, 3))
...         return transpose4
>>>
>>> class OuterBlock(nn.Cell):
...     @lazy_inline
...     def __init__(self):
...         super(OuterBlock, self).__init__()
...         self.block = Block()
...     def construct(self, x):
...         return self.block(x)
>>>
>>> class Nets(nn.Cell):
...     def __init__(self):
...         super(Nets, self).__init__()
...         self.blocks = nn.CellList()
...         for _ in range(3):
...             b = OuterBlock()
...             b.offload()
...             self.blocks.append(b)
...     def construct(self, x):
...         out = x
...         for i in range(3):
...             out = self.blocks[i](out)
...         return out
```

property param_prefix

Param prefix is the prefix of current cell's direct child parameter.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dense = nn.Dense(2, 2)
...
...     def construct(self, x):
...         x = self.dense(x)
...         return x
>>> net = Net()
>>> net.update_cell_prefix()
>>> print(net.dense.param_prefix)
```

(continues on next page)

(continued from previous page)

```
dense
```

property parameter_layout_dict

parameter_layout_dict represents the tensor layout of a parameter, which is inferred by shard strategy and distributed operator information.

parameters_and_names(*name_prefix=''*, *expand=True*)

Returns an iterator over cell parameters.

Includes the parameter's name and itself.

Parameters

- **name_prefix** (*str*) –Namespace. Default: `''` .
- **expand** (*bool*) –If true, yields parameters of this cell and all subcells. Otherwise, only yield parameters that are direct members of this cell. Default: `True` .

Returns

Iteration, all the names and corresponding parameters in the cell.

Examples

```
>>> from mindspore import nn
>>> n = nn.Dense(3, 4)
>>> names = []
>>> for m in n.parameters_and_names():
...     if m[0]:
...         names.append(m[0])
```

Tutorial Examples:

- Building a Network - Model Parameters

parameters_broadcast_dict(*recurse=True*)

Gets the parameters broadcast dictionary of this cell.

Parameters

recurse (*bool*) –Whether contains the parameters of subcells. Default: `True` .

Returns

OrderedDict, return parameters broadcast dictionary.

parameters_dict(*recurse=True*)

Gets the parameters dictionary of this cell.

Parameters

recurse (*bool*) –Whether contains the parameters of subcells. Default: `True` .

Returns

OrderedDict, return parameters dictionary.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, Parameter
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dense = nn.Dense(2, 2)
...
...     def construct(self, x):
...         x = self.dense(x)
...         return x
>>> net = Net()
>>> print(net.parameters_dict())
OrderedDict([('dense.weight', Parameter(name=dense.weight, shape=(2, 2), dtype=Float32,
requires_grad=True)), ('dense.bias', Parameter(name=dense.bias, shape=(2,),␣
↪dtype=Float32,
requires_grad=True))])
```

property pipeline_stage

pipeline_stage represents the pipeline stage of current Cell.

place(*role*, *rank_id*)

Set the label for all operators in this cell. This label tells MindSpore compiler on which process this cell should be launched. And each process's identical label consists of input *role* and *rank_id*. So by setting different cells with different labels, which will be launched on different processes, users can launch a distributed training or predicting job.

Note:

- This method is effective only after *mindspore.communication.init()* is called for dynamic cluster building.

Parameters

- **role** (*str*) – The role of the process on which this cell will be launched. Only 'MS_WORKER' is supported for now.
- **rank_id** (*int*) – The rank id of the process on which this cell will be launched. The rank is unique in processes with the same role.

Examples

```
>>> from mindspore import context
>>> import mindspore.nn as nn
>>> context.set_context(mode=context.GRAPH_MODE)
>>> fc = nn.Dense(2, 3)
>>> fc.place('MS_WORKER', 0)
```

recompute(***kwargs*)

Set the cell recomputed. All the primitive in the cell except the outputs will be set recomputed. If a primitive set recomputed feeds into some backward nodes for computing gradient, rather than storing the intermediate activation computed in forward pass, we will recompute it in backward pass.

Note:

- If the computation involves something like randomization or global variable, the equivalence is not guaranteed currently.
- If the recompute api of a primitive in this cell is also called, the recompute mode of this primitive is subject to the recompute api of the primitive.
- The interface can be configured only once. Therefore, when the parent cell is configured, the child cell should not be configured.
- The outputs of cell are excluded from recomputation by default, which is based on our configuration experience to reduce memory footprint. If a cell has only one primitive and the primitive is wanted to be set recomputed, use the recompute api of the primtive.
- When the memory remains after applying the recomputation, configuring 'mp_comm_recompute=False' to improve performance if necessary.
- When the memory still not enough after applying the recompute, configuring 'parallel_optimizer_comm_recompute=True' to save more memory if necessary. Cells in the same fusion group should have the same parallel_optimizer_comm_recompute configures.

Parameters

- **mp_comm_recompute** (*bool*) −Specifies whether the model parallel communication operators in the cell are recomputed in auto parallel or semi auto parallel mode. Default: `True` .
- **parallel_optimizer_comm_recompute** (*bool*) −Specifies whether the communication operator allgathers introduced by optimizer shard are recomputed in auto parallel or semi auto parallel mode. Default: `False` .

register_backward_hook(*hook_fn*)

Register the backward hook function.

Note:

- The *register_backward_hook(hook_fn)* does not work in graph mode or functions decorated with 'jit'.
- The 'hook_fn' must be defined as the following code. *cell* is the registered Cell object. *grad_input* is the gradient computed and passed to the next Cell or primitive, which can be return a new gradient or None. *grad_output* is the gradient passed to the Cell.
- The 'hook_fn' should have the following signature: hook_fn(cell, grad_input, grad_output) -> New grad_input gradient or none.
- The 'hook_fn' is executed in the python environment. In order to prevent running failed when switching to graph mode, it is not recommended to write it in the *construct* function of Cell object. In the pynative mode, if the *register_backward_hook* function is called in the *construct* function of the Cell object, a hook function will be added at each run time of Cell object.

Parameters

hook_fn (*function*) −Python function. Backward hook function.

Returns

A handle corresponding to the *hook_fn* . The handle can be used to remove the added *hook_fn* by calling *handle.remove()* .

Raises

TypeError −If the *hook_fn* is not a function of python.

Supported Platforms: `Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def backward_hook_fn(cell, grad_input, grad_output):
...     print("backward input: ", grad_output)
...     print("backward output: ", grad_input)
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...         self.handle = self.relu.register_backward_hook(backward_hook_fn)
...
...     def construct(self, x):
...         x = x + x
...         x = self.relu(x)
...         return x
>>> grad = ops.GradOperation(get_all=True)
>>> net = Net()
>>> output = grad(net)(Tensor(np.ones([1]).astype(np.float32)))
backward input: (Tensor(shape=[1], dtype=Float32, value= [ 1.00000000e+00]),)
backward output: (Tensor(shape=[1], dtype=Float32, value= [ 1.00000000e+00]),)
>>> print(output)
(Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]),)
```

register_backward_pre_hook(*hook_fn*)

Register the backward pre hook function.

Note:

- The *register_backward_pre_hook(hook_fn)* does not work in graph mode or functions decorated with 'jit'.
- The 'hook_fn' must be defined as the following code. *cell* is the Cell object. *grad_output* is the gradient passed to the Cell.
- The 'hook_fn' should have the following signature: hook_fn(cell, grad_output) -> New grad_output gradient or None.
- The 'hook_fn' is executed in the python environment. In order to prevent running failed when switching to graph mode, it is not recommended to write it in the *construct* function of Cell object.
- In the pynative mode, if the *register_backward_pre_hook* function is called in the *construct* function of the Cell object, a hook function will be added at each run time of Cell object.

Parameters

hook_fn (*function*) –Python function. Backward pre hook function.

Returns

A handle corresponding to the *hook_fn* . The handle can be used to remove the added *hook_fn* by calling *handle.remove()* .

Raises

TypeError –If the *hook_fn* is not a function of python.

Supported Platforms: `Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def backward_pre_hook_fn(cell, grad_output):
...     print("backward input: ", grad_output)
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...         self.handle = self.relu.register_backward_pre_hook(backward_pre_hook_fn)
...
...     def construct(self, x):
...         x = x + x
...         x = self.relu(x)
...         return x
>>> grad = ops.GradOperation(get_all=True)
>>> net = Net()
>>> output = grad(net)(Tensor(np.ones([1]).astype(np.float32)))
backward input: (Tensor(shape=[1], dtype=Float32, value= [ 1.00000000e+00]),)
>>> print(output)
(Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]),)
```

register_buffer(*name: str*, *tensor: Optional[*Tensor*]*, *persistent: bool = True*)

Add a buffer to the cell.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's *running_mean* is not a parameter, but is part of the cell's state. Buffers, by default, are persistent and will be saved alongside parameters. This behavior can be changed by setting *persistent* to `False` . The only difference between a persistent buffer and a non-persistent buffer is that the latter will not be a part of this cell's *state_dict* .

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*str*) –name of the buffer. The buffer can be accessed from this cell using the given name.
- **tensor** (`Tensor`) –Buffer to be registered. If `None` , the buffer is not included in the cell's *state_dict* .
- **persistent** (*bool, optional*) –Whether the buffer is part of this cell's *state_dict*. Default `True`.

Examples

```
>>> import mindspore
...
>>> class Net(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer0", mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.net_buffer
...
>>> net = Net()
>>> net.register_buffer("buffer0", mindspore.tensor([4, 5, 6]))
>>> print(net.buffer0)
[4 5 6]
```

register_forward_hook(*hook_fn*)

Set the Cell forward hook function.

Note:

- The *register_forward_hook(hook_fn)* does not work in graph mode or functions decorated with 'jit'.
- 'hook_fn' must be defined as the following code. *cell* is the object of registered Cell. *inputs* is the forward input objects passed to the Cell. *output* is the forward output object of the Cell. The 'hook_fn' can modify the forward output object by returning new forward output object.
- It should have the following signature: hook_fn(cell, inputs, output) -> new output object or none.
- In order to prevent running failed when switching to graph mode, it is not recommended to write it in the *construct* function of Cell object. In the pynative mode, if the *register_forward_hook* function is called in the *construct* function of the Cell object, a hook function will be added at each run time of Cell object.

Parameters

hook_fn (*function*) – Python function. Forward hook function.

Returns

A handle corresponding to the *hook_fn* . The handle can be used to remove the added *hook_fn* by calling *handle.remove()* .

Raises

TypeError – If the *hook_fn* is not a function of python.

Supported Platforms: `Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def forward_hook_fn(cell, inputs, output):
...     print("forward inputs: ", inputs)
...     print("forward output: ", output)
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.mul = nn.MatMul()
...         self.handle = self.mul.register_forward_hook(forward_hook_fn)
...
...     def construct(self, x, y):
...         x = x + x
...         x = self.mul(x, y)
...         return x
>>> grad = ops.GradOperation(get_all=True)
>>> net = Net()
>>> output = grad(net)(Tensor(np.ones([1]).astype(np.float32)), Tensor(np.ones([1]).
↪astype(np.float32)))
forward inputs: (Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]),␣
↪Tensor(shape=[1],
                dtype=Float32, value= [ 1.00000000e+00]))
forward output: 2.0
>>> print(output)
(Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]), Tensor(shape=[1],␣
↪dtype=Float32,
value= [ 2.00000000e+00]))
```

register_forward_pre_hook(*hook_fn*)

Register forward pre hook function for Cell object.

Note:

- The *register_forward_pre_hook(hook_fn)* does not work in graph mode or functions decorated with 'jit'.
- 'hook_fn' must be defined as the following code. *cell* is the object of registered Cell. *inputs* is the forward input objects passed to the Cell. The 'hook_fn' can modify the forward input objects by returning new forward input objects.
- It should have the following signature: hook_fn(cell, inputs) -> new input objects or none.
- In order to prevent running failed when switching to graph mode, it is not recommended to write it in the *construct* function of Cell object. In the pynative mode, if the *register_forward_pre_hook* function is called in the *construct* function of the Cell object, a hook function will be added at each run time of Cell object.

Parameters

hook_fn (*function*) –Python function. Forward pre hook function.

Returns

A handle corresponding to the *hook_fn* . The handle can be used to remove the added *hook_fn* by calling *handle.remove()* .

Raises

TypeError – If the *hook_fn* is not a function of python.

Supported Platforms: `Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def forward_pre_hook_fn(cell, inputs):
...     print("forward inputs: ", inputs)
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.mul = nn.MatMul()
...         self.handle = self.mul.register_forward_pre_hook(forward_pre_hook_fn)
...
...     def construct(self, x, y):
...         x = x + x
...         x = self.mul(x, y)
...         return x
>>> grad = ops.GradOperation(get_all=True)
>>> net = Net()
>>> output = grad(net)(Tensor(np.ones([1]).astype(np.float32)), Tensor(np.ones([1]).
↪astype(np.float32)))
forward inputs: (Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]),␣
↪Tensor(shape=[1],
               dtype=Float32, value= [ 1.00000000e+00]))
>>> print(output)
(Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00]), Tensor(shape=[1],␣
↪dtype=Float32,
value= [ 2.00000000e+00]))
```

register_load_state_dict_post_hook(*hook*)

Register a post-hook to be run after cell's *mindspore.nn.Cell.load_state_dict()* is called.

It should have the following signature:

hook(cell, incompatible_keys) -> None

The `cell` argument is the current cell that this hook is registered on, and the `incompatible_keys` argument is a `NamedTuple` consisting of attributes `missing_keys` and `unexpected_keys`. `missing_keys` is a `list` of `str` containing the missing keys and `unexpected_keys` is a `list` of `str` containing the unexpected keys.

The given incompatible_keys can be modified inplace if needed.

Note that the checks performed when calling *load_state_dict()* with `strict=True` are affected by modifications the hook makes to `missing_keys` or `unexpected_keys`, as expected. Additions to either set of keys will result in an error being thrown when `strict=True`, and clearing out both missing and unexpected keys will avoid an error.

Parameters

hook (*Callable*) – The hook function after *load_state_dict* is called.

Returns

A handle that can be used to remove the added hook by calling *handle.remove()*.

register_load_state_dict_pre_hook(*hook*)
Register a pre-hook to be run before cell's *mindspore.nn.Cell.load_state_dict()* is called.

It should have the following signature:

hook(cell, state_dict, prefix, local_metadata, strict, missing_keys, unexpected_keys, error_msgs) -> None # noqa: B950

Parameters

hook (*Callable*) –The hook function before *load_state_dict* is called.

Returns

A handle that can be used to remove the added hook by calling *handle.remove()*.

register_state_dict_post_hook(*hook*)
Register a post-hook for the *mindspore.nn.Cell.state_dict()* method.

It should have the following signature:

hook(cell, state_dict, prefix, local_metadata) -> None

The registered hooks can modify the `state_dict` inplace.

Parameters

hook (*Callable*) –The hook function after *state_dict* is called.

Returns

A handle that can be used to remove the added hook by calling *handle.remove()*.

register_state_dict_pre_hook(*hook*)
Register a pre-hook for the *mindspore.nn.Cell.state_dict()* method.

It should have the following signature:

hook(cell, prefix, keep_vars) -> None

The registered hooks can be used to perform pre-processing before the *state_dict* call is made.

Parameters

hook (*Callable*) –The hook function before *state_dict* is called.

Returns

A handle that can be used to remove the added hook by calling *handle.remove()*.

Examples

```
>>> import mindspore
...
...
>>> class NetA(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_a", mindspore.tensor([1, 2, 3]))
...         self.param_a = mindspore.Parameter(mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.buffer_a + self.param_a
...
...
```

(continues on next page)

(continued from previous page)

```
>>> def _add_extra_param(cell, prefix, keep_vars):
...     cell._params["extra_param"] = mindspore.Parameter(mindspore.tensor([4, 5, 6]))
...
...
>>> net = NetA()
>>> handle = net.register_state_dict_pre_hook(_add_extra_param)
>>> net_state_dict = net.state_dict()
>>> handle.remove()
>>> print("extra_param" in net_state_dict)
True
```

remove_redundant_parameters()
Remove the redundant parameters.

This interface usually needs not to be used explicitly.

run_construct(*cast_inputs*, *kwargs*)
Run the construct function.

Note: This function will be removed in a future version. It is not recommended to call this function.

Parameters

- **cast_inputs** (*tuple*) –The input objects of Cell.
- **kwargs** (*dict*) –Provide keyword arguments.

Returns

output, the output object of Cell.

set_boost(*boost_type*)
In order to improve the network performance, configure the network auto enable to accelerate the algorithm in the algorithm library.

If *boost_type* is not in the algorithm library, please view the algorithm in the algorithm library through algorithm library.

Note: Some acceleration algorithms may affect the accuracy of the network, please choose carefully.

Parameters

boost_type (*str*) –accelerate algorithm.

Returns

Cell, the cell itself.

Raises

ValueError –If boost_type is not in the algorithm library.

set_broadcast_flag(*mode=True*)
Set parameter broadcast mode for this cell.

Parameters

mode (*bool*) –Specifies whether the mode is parameter broadcast. Default: `True` .

set_comm_fusion(*fusion_type*, *recurse=True*)

Set *comm_fusion* for all the parameters in this cell. Please refer to the description of *mindspore.Parameter.comm_fusion*.

Note: The value of attribute will be overwritten when the function is called multiply.

Parameters

- **fusion_type** (*int*) −The value of *comm_fusion*.
- **recurse** (*bool*) −Whether sets the trainable parameters of subcells. Default: `True` .

set_data_parallel()

For all primitive ops in this cell(including ops of cells that wrapped by this cell), if parallel strategy is not specified, then instead of auto-searching, data parallel strategy will be generated for those primitive ops.

Note: Only effective while using auto_parallel_context = ParallelMode.AUTO_PARALLEL under graph mode.

Examples

```
>>> import mindspore.nn as nn
>>> net = nn.Dense(3, 4)
>>> net.set_data_parallel()
```

set_extra_state(*state: Any*)

Set extra state contained in the loaded *state_dict*.

This function is called from *load_state_dict* to handle any extra state found within the *state_dict*. Implement this function and a corresponding *get_extra_state* for your cell if you need to store extra state within its *state_dict*.

Parameters

state (*dict*) −Extra state from the *state_dict*.

set_grad(*requires_grad=True*)

Sets the cell flag for gradient.

Parameters

requires_grad (*bool*) −Specifies if the net need to grad, if it is `true` , the cell will construct backward network in pynative mode. Default: `True` .

Returns

Cell, the cell itself.

set_inputs(**inputs*, ***kwargs*)

Save set inputs for computation graph. The number of inputs should be the same with that of the datasets. When using Model for dynamic shape, please make sure that all networks and loss functions passed to the Model are configured with set_inputs. The shape of input Tensor can be either dynamic or static.

Note: There are two mode:

- Full mode: arguments will be used as all compile inputs for graph-compiling.

- Incremental mode: arguments will set to some of the Cell inputs, which will be substituted into the input at the corresponding position for graph-compiling.

Only one of inputs or kwargs can be set. Inputs for full mode and kwargs for incremental mode.

Parameters

- **inputs** (*tuple*) –Full mode arguments.
- **kwargs** (*dict*) –Incremental mode arguments. The acceptable key is the name of parameter defined in *self.construct*.

Warning: This is an experimental API that is subject to change or deletion.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, Tensor
>>>
>>> class ReluNet(nn.Cell):
...     def __init__(self):
...         super(ReluNet, self).__init__()
...         self.relu = nn.ReLU()
...     def construct(self, x):
...         return self.relu(x)
>>>
>>> net = ReluNet()
>>> input_dyn = Tensor(shape=[3, None], dtype=ms.float32)
>>> net.set_inputs(input_dyn)
>>> input = Tensor(np.random.random([3, 10]), dtype=ms.float32)
>>> output = net(input)
>>>
>>> net2 = ReluNet()
>>> net2.set_inputs(x=input_dyn)
>>> output = net2(input)
```

set_jit_config(*jit_config*)

Set jit config for cell.

Parameters

jit_config (JitConfig) –Jit config for compile. For details, please refer to *mindspore.JitConfig*.

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
...
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.relu = nn.ReLU()
...
...     def construct(self, x):
...         x = self.relu(x)
...         return x
>>> net = Net()
>>> jitconfig = ms.JitConfig()
>>> net.set_jit_config(jitconfig)
```

set_param_ps(*recurse=True*, *init_in_server=False*)

Set whether the trainable parameters are updated by parameter server and whether the trainable parameters are initialized on server.

Note: It only works when a running task is in the parameter server mode. It is only supported in graph mode.

Parameters

- **recurse** (*bool*) –Whether sets the trainable parameters of subcells. Default: `True` .
- **init_in_server** (*bool*) –Whether trainable parameters updated by parameter server are initialized on server. Default: `False` .

set_train(*mode=True*)

Sets the cell to training mode.

The cell itself and all children cells will be set to training mode. Layers that have different constructions for training and predicting, such as *BatchNorm*, will distinguish between the branches by this attribute. If set to true, the training branch will be executed, otherwise another branch.

Note: When execute function Model.train(), framework will call Cell.set_train(True). When execute function Model.eval(), framework will call Cell.set_train(False).

Parameters

mode (*bool*) –Specifies whether the model is training. Default: `True` .

Returns

Cell, the cell itself.

Tutorial Examples:

- Model Training - Implementing Training and Evaluation

shard(*in_strategy*, *out_strategy=None*, *parameter_plan=None*, *device='Ascend'*, *level=0*)

Defining the input and output layouts of this cell and the parallel strategies of remaining ops will be generated by sharding propagation. In PyNative mode, use this method to specify a Cell for distributed execution in graph mode. In Graph mode,

use this method to specify distribution strategy for a Cell, strategy for others will be set by sharding propagation. in_strategy and out_strategy define the input and output layout respectively. in_strategy/out_strategy should be a tuple, each element of which corresponds to the desired layout of this input/output, which can refer to the description of *mindspore.ops.Primitive.shard()*. The parallel strategies of remaining operators are derived from the strategy specified by the input and output.

Note:

- It is valid only in semi auto parallel or auto parallel mode. In other parallel modes, strategies set here will be ignored.
- If the input contain Parameter, its strategy should be set in *in_strategy*.

Parameters

- **in_strategy** (*tuple*) –Define the layout of inputs, each element of the tuple should be a tuple. Tuple defines the layout of the corresponding input.
- **out_strategy** (*Union[None, tuple]*) –Define the layout of outputs similar with in_strategy. It is not in use right now. Default: `None` .
- **parameter_plan** (*Union[dict, None]*) –Define the layout for the specified parameters. Each element in dict defines the layout of the parameter like "param_name: layout". The key is a parameter name of type 'str'. The value is a 1-D integer tuple, indicating the corresponding layout. If the parameter name is incorrect or the corresponding parameter has been set, the parameter setting will be ignored. Default: `None` .
- **device** (*str*) –Select a certain device target. It is not in use right now. Support [`"CPU"` , `"GPU"` , `"Ascend"`]. Default: `"Ascend"` .
- **level** (*int*) –Option for parallel strategy infer algorithm, namely the object function, maximize computation over communication ratio, maximize speed performance, minimize memory usage etc. It is not in use right now. Support [`"0"` , `"1"` , `"2"`]. Default: `0` .

Returns

Function, return the cell construct function that will be executed under auto parallel process.

Examples

```
>>> import mindspore.nn as nn
>>>
>>> class Block(nn.Cell):
...   def __init__(self):
...     self.dense1 = nn.Dense(10, 10)
...     self.relu = nn.ReLU()
...     self.dense2 = nn.Dense2(10, 10)
...   def construct(self, x):
...     x = self.relu(self.dense2(self.relu(self.dense1(x))))
...     return x
>>>
>>> class example(nn.Cell):
...   def __init__(self):
...     self.block1 = Block()
...     self.block2 = Block()
...     self.block2_shard = self.block2.shard(in_strategy=((2, 1),),
...                                           parameter_plan={'self.block2.shard.dense1.
↪weight': (4, 1)})
...   def construct(self, x):
```

(continues on next page)

(continued from previous page)

```
...     x = self.block1(x)
...     x = self.block2_shard(x)
...     return x
```

state_dict (**args*, *destination=None*, *prefix=''*, *keep_vars=False*)

Return a dictionary containing references to the whole state of the cell.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names. Parameters and buffers set to `None` are not included.

Note: The returned object is a shallow copy. It contains references to the cell's parameters and buffers.

Warning:

- Currently `state_dict()` also accepts positional arguments for `destination`, `prefix` and `keep_vars` in order. However, this is being deprecated and keyword arguments will be enforced in future releases.
- Please avoid the use of argument `destination` as it is not designed for end-users.

Parameters

- **destination** (*dict, optional*) – If provided, the state of cell will be updated into the dict and the same object is returned. Otherwise, an `OrderedDict` will be created and returned. Default: `None`.
- **prefix** (*str, optional*) – A prefix added to parameter and buffer names to compose the keys in state_dict. Default: `''`.
- **keep_vars** (*bool, optional*) – Whether the state_dict returns a copy. Default: `False` , returns a reference.

Returns

Dict, a dictionary containing a whole state of the cell.

Examples

```
>>> import mindspore
>>> class Model(mindspore.nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.register_buffer("buffer_a", mindspore.tensor([4, 5, 6]))
...         self.param_a = mindspore.Parameter(mindspore.tensor([1, 2, 3]))
...
...     def construct(self, x):
...         return x + self.buffer_a + self.param_a
...
...
>>> model = Model()
>>> print(model.state_dict())
OrderedDict([('param_a', Parameter (name=param_a, shape=(3,), dtype=Int64, requires_
↪grad=True)), \
('buffer_a', Tensor(shape=[3], dtype=Int64, value= [4, 5, 6]))])
```

to_float (*dst_type*)

Add cast on all inputs of cell and child cells to run with certain float type.

If *dst_type* is *mindspore.dtype.float16*, all the inputs of Cell, including input, Parameter and Tensor, will be cast to float16. Please refer to the usage in source code of `mindspore.amp.build_train_network()`.

Note: Multiple calls will overwrite.

Parameters

dst_type (*mindspore.dtype*) −Transfer cell to run with dst_type. dst_type can be *mstype.float16* , *mstype.float32* or *mstype.bfloat16*.

Returns

Cell, the cell itself.

Raises

ValueError −If dst_type is not *mstype.float32* , *mstype.float16* or *mstype.bfloat16*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore import dtype as mstype
>>>
>>> net = nn.Conv2d(120, 240, 4, has_bias=False, weight_init='normal')
>>> net.to_float(mstype.float16)
Conv2d(input_channels=120, output_channels=240, kernel_size=(4, 4), stride=(1, 1), pad_
↪mode=same,
padding=0, dilation=(1, 1), group=1, has_bias=False, weight_init=normal, bias_init=None,␣
↪format=NCHW)
```

trainable_params(*recurse=True*)

Returns all trainable parameters.

Returns a list of all trainable parameters.

Parameters

recurse (*bool*) −Whether contains the trainable parameters of subcells. Default: `True` .

Returns

List, the list of trainable parameters.

Tutorial Examples:

- Model Training - Optimizer

untrainable_params(*recurse=True*)

Returns all untrainable parameters.

Returns a list of all untrainable parameters.

Parameters

recurse (*bool*) −Whether contains the untrainable parameters of subcells. Default: `True` .

Returns

List, the list of untrainable parameters.

update_cell_prefix()
Update the *param_prefix* of all child cells.

After being invoked, it can get all the cell's children's name prefix by '_param_prefix'.

update_cell_type(*cell_type*)
The current cell type is updated when a quantization aware training network is encountered.

After being invoked, it can set the cell type to 'cell_type'.

Parameters

cell_type (*str*) –The type of cell to be updated, cell_type can be "quant" or "second-order".

update_parameters_name(*prefix=''*, *recurse=True*)
Adds the *prefix* string to the names of parameters.

Parameters

- **prefix** (*str*) –The prefix string. Default: `''` .
- **recurse** (*bool*) –Whether contains the parameters of subcells. Default: `True` .

3.1.2 mindspore.nn.GraphCell

class mindspore.nn.**GraphCell**(*graph*, *params_init=None*, *obf_random_seed=None*)
Base class for running the graph loaded from a MindIR.

This feature is still under development. Currently *GraphCell* do not support modifying the structure of the diagram, and can only use data that shape and type are the same as the input when exporting the MindIR.

Parameters

- **graph** (*FuncGraph*) –A compiled graph loaded from MindIR.
- **params_init** (*dict*) –Parameters need to be inited in the graph. The key is the parameter name whose type is str, and the value is a Tensor or Parameter. If the parameter exists in the graph according to the name, update it's value. If the parameter does not exist, ignore it. Default: `None` .
- **obf_random_seed** (*Union[int, None]*) –The random seed used for dynamic obfuscation, which is not supported now.

Raises

- **NotImplementedError** –Dynamic structure obfuscation is not supported now.
- **TypeError** –If the *graph* is not a FuncGraph.
- **TypeError** –If the *params_init* is not a dict.
- **TypeError** –If the key of the *params_init* is not a str.
- **TypeError** –If the value of the *params_init* is neither a Tensor nor a Parameter.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore import context
>>> context.set_context(mode=context.GRAPH_MODE)
>>> net = nn.Conv2d(1, 1, kernel_size=3, weight_init="ones")
>>> input = Tensor(np.ones([1, 1, 3, 3]).astype(np.float32))
>>> ms.export(net, input, file_name="net", file_format="MINDIR")
>>> graph = ms.load("net.mindir")
>>> net = nn.GraphCell(graph)
>>> output = net(input)
>>> print(output)
[[[[4. 6. 4.]
   [6. 9. 6.]
   [4. 6. 4.]]]]
```

3.1.3 mindspore.nn.LossBase

class mindspore.nn.**LossBase**(*reduction='mean'*)

Base class for other losses.

Other losses derived from this should implement their own *construct* and use method *self.get_loss* to apply reduction to loss values.

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the (weighted) mean of elements in the output.
- `'sum'`: the output elements will be summed.

Raises

ValueError – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import ops, Tensor, nn
>>> import numpy as np
>>>
>>> class Net(nn.LossBase):
...     def __init__(self, reduction='mean'):
...         super(Net, self).__init__(reduction)
...         self.abs = ops.Abs()
...
...     def construct(self, logits, labels):
```

(continues on next page)

(continued from previous page)

```
...         x = self.abs(logits - labels)
...         output = self.get_loss(x)
...         axis = self.get_axis(x)
...         return output, axis
>>> net = Net()
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output, axis = net(logits, labels)
>>> print(output)
0.33333334
>>> print(axis)
(0,)
>>> # Case 2: logits.shape = labels.shape = (3, 3)
>>> logits = Tensor(np.array([[1, 2, 3],[1, 2, 3],[1, 2, 3]]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 2, 2],[1, 2, 3],[1, 2, 3]]), mindspore.float32)
>>> output, axis = net(logits, labels)
>>> print(output)
0.11111111
>>> print(axis)
(0, 1)
```

get_axis(*x*)

Get a range of axis for input.

Parameters

x (Tensor) – Tensor of any shape.

get_loss(*x*, *weights=1.0*)

Computes the weighted loss.

Parameters

- **x** (Tensor) – Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **weights** (*Union[float, Tensor], optional*) – Weights. When *weights* is a Tensor, the rank is either 0, or the same rank as inputs, and must be broadcastable to inputs (i.e., all dimensions must be either *1*, or the same as the corresponding inputs dimension). Default: `1.0`.

Returns

Return the weighted loss.

3.1.4 mindspore.nn.Optimizer

class mindspore.nn.**Optimizer**(*learning_rate*, *parameters*, *weight_decay=0.0*, *loss_scale=1.0*)

Base class for updating parameters. Never use this class directly, but instantiate one of its subclasses instead.

Grouping parameters is supported. If parameters are grouped, different strategy of *learning_rate*, *weight_decay* and *grad_centralization* can be applied to each group.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule]*) –
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **parameters** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *parameters* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **weight_decay** (*Union[float, int], optional*) –An int or a floating point value for the weight decay. It must be equal to or greater than 0. If the type of *weight_decay* input is int, it will be converted to float. Default: `0.0` .
- **loss_scale** (*float, optional*) –A floating point value for the loss scale. It must be greater than 0. If the type of *loss_scale* input is int, it will be converted to float. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False` , this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0` .

Raises

- **TypeError** –If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** –If element of *parameters* is neither Parameter nor dict.
- **TypeError** –If *loss_scale* is not a float.
- **TypeError** –If *weight_decay* is neither float nor int.
- **ValueError** –If *loss_scale* is less than or equal to 0.
- **ValueError** –If *weight_decay* is less than 0.
- **ValueError** –If *learning_rate* is a Tensor, but the dimension of tensor is greater than 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, ops, Tensor
>>>
>>> class MyMomentum(nn.Optimizer):
...     def __init__(self, params, learning_rate, momentum=0.9):
...         super(MyMomentum, self).__init__(learning_rate, params)
...         self.moments = self.parameters.clone(prefix="moments", init="zeros")
...         self.momentum = momentum
...         self.opt = ops.ApplyMomentum()
...
...     def construct(self, gradients):
...         params = self.parameters
...         lr = self.get_lr()
...         gradients = self.flatten_gradients(gradients)
...         gradients = self.decay_weight(gradients)
...         gradients = self.gradients_centralization(gradients)
...         gradients = self.scale_grad(gradients)
...
...         success = None
...         for param, mom, grad in zip(params, self.moments, gradients):
...             success = self.opt(param, mom, lr, grad, self.momentum)
...         return success
>>>
>>> net = nn.Dense(2, 3)
>>> loss_fn = nn.MAELoss()
>>> opt = MyMomentum(net.trainable_params(), 0.01)
>>>
>>> device_target = opt.target
>>> opt_unique = opt.unique
>>> weight_decay_value = opt.get_weight_decay()
>>>
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>>
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, opt.parameters, has_aux=True)
>>>
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     opt(grads)
...     return loss
>>>
>>> data = Tensor(np.random.rand(4, 10, 2), mindspore.dtype.float32)
>>> label = Tensor(np.random.rand(4, 10, 3), mindspore.dtype.float32)
>>> train_step(data, label)
```

broadcast_params(*optim_result*)

Apply Broadcast operations in the sequential order of parameter groups.

Parameters

optim_result (*bool*) – The results of updating parameters. This input is used to ensure that the parameters are updated before they are broadcast.

Returns

The broadcast parameters.

decay_weight(*gradients*)

Weight decay.

An approach to reduce the overfitting of a deep learning neural network model. User-defined optimizers based on *mindspore.nn.Optimizer* can also call this interface to apply weight decay.

Parameters

gradients (*tuple[Tensor]*) – The gradients of network parameters, and have the same shape as the parameters.

Returns

tuple[Tensor], The gradients after weight decay.

flatten_gradients(*gradients*)

Flatten gradients into several chunk tensors grouped by data type if network parameters are flattened.

A method to enable performance improvement by using contiguous memory for parameters and gradients. User-defined optimizers based on *mindspore.nn.Optimizer* should call this interface to support contiguous memory for network parameters.

Parameters

gradients (*tuple[Tensor]*) – The gradients of network parameters.

Returns

tuple[Tensor], The gradients after flattened, or the original gradients if parameters are not flattened.

get_lr()

The optimizer calls this interface to get the learning rate for the current step. User-defined optimizers based on *mindspore.nn.Optimizer* can also call this interface before updating the parameters.

Returns

float, the learning rate of current step.

get_lr_parameter(*param*)

When parameters is grouped and learning rate is different for each group, get the learning rate of the specified *param*.

Parameters

param (*Union[Parameter, list[Parameter]]*) – The *Parameter* or list of *Parameter*.

Returns

A single *Parameter* or *list[Parameter]* according to the input type. If learning rate is dynamic, *LearningRateSchedule* or *list[LearningRateSchedule]* that used to calculate the learning rate will be returned.

Examples

```
>>> from mindspore import nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'lr': 0.05},
...                 {'params': no_conv_params, 'lr': 0.01}]
```

(continues on next page)

(continued from previous page)

```
>>> optim = nn.Momentum(group_params, learning_rate=0.1, momentum=0.9, weight_decay=0.0)
>>> conv_lr = optim.get_lr_parameter(conv_params)
>>> print(conv_lr[0].asnumpy())
0.05
```

get_weight_decay()

The optimizer calls this interface to get the weight decay value for the current step. User-defined optimizers based on *mindspore.nn.Optimizer* can also call this interface before updating the parameters.

Returns

float, the weight decay value of current step.

gradients_centralization(*gradients*)

Gradients centralization.

A method for optimizing convolutional layer parameters to improve the training speed of a deep learning neural network model. User-defined optimizers based on *mindspore.nn.Optimizer* can also call this interface to centralize gradients.

Parameters

gradients (*tuple[Tensor]*) – The gradients of network parameters, and have the same shape as the parameters.

Returns

tuple[Tensor], The gradients after gradients centralization.

scale_grad(*gradients*)

Restore gradients for mixed precision.

User-defined optimizers based on *mindspore.nn.Optimizer* can also call this interface to restore gradients.

Parameters

gradients (*tuple[Tensor]*) – The gradients of network parameters, and have the same shape as the parameters.

Returns

tuple[Tensor], The gradients after loss scale.

property target

The property is used to determine whether the parameter is updated on host or device. The input type is str and can only be 'CPU', 'Ascend' or 'GPU'.

property unique

Whether to make the gradients unique in optimizer. Generally, it is used in sparse networks. Set to True if the gradients of the optimizer are sparse, while set to False if the forward network has made the parameters unique, that is, the gradients of the optimizer is no longer sparse. The default value is True when it is not set.

3.2 Container

API Name	Description	Supported Platforms
mindspore.nn.CellDict	Holds Cells in a dictionary.	Ascend GPU CPU
mindspore.nn.CellList	Holds Cells in a list.	Ascend GPU CPU
mindspore.nn.SequentialCell	Sequential Cell container.	Ascend GPU CPU

3.2.1 mindspore.nn.CellDict

class mindspore.nn.**CellDict**(**args*, ***kwargs*)

Holds Cells in a dictionary. For more details about *Cell* , please refer to *mindspore.nn.Cell* .

CellDict can be used like a regular Python dictionary.

Parameters

- **args** (*iterable, optional*) – An iterable of key-value pairs of (key, Cell), the type of key-value pairs is (string, Cell); Or a mapping(dictionary) from string to Cell. The type of Cell can not be CellDict, CellList or SequentialCell. The key can not be same with the attributes of class Cell, can not contain '.', can not be an empty string. The key of type string is used to search corresponding Cell in the CellDict.
- **kwargs** (*dict*) – Reserved for keyword argument to be expanded.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import collections
>>> from collections import OrderedDict
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, nn
>>>
>>> cell_dict = nn.CellDict({'conv': nn.Conv2d(10, 6, 5),
...                          'relu': nn.ReLU(),
...                          'max_pool2d': nn.MaxPool2d(kernel_size=4, stride=4)})
>>> print(len(cell_dict))
3
>>> cell_dict.clear()
>>> print(len(cell_dict))
0
>>> ordered_cells = OrderedDict([('conv', nn.Conv2d(10, 6, 5, pad_mode='valid')),
...                              ('relu', nn.ReLU()),
...                              ('max_pool2d', nn.MaxPool2d(kernel_size=2, stride=2))])
>>> cell_dict.update(ordered_cells)
>>> x = Tensor(np.ones([1, 10, 6, 10]), ms.float32)
>>> for cell in cell_dict.values():
...     x = cell(x)
>>> print(x.shape)
(1, 6, 1, 3)
>>> x = Tensor(np.ones([1, 10, 6, 10]), ms.float32)
>>> for item in cell_dict.items():
...     x = item[1](x)
>>> print(x.shape)
(1, 6, 1, 3)
>>> print(cell_dict.keys())
odict_keys(['conv', 'relu', 'max_pool2d'])
>>> pop_cell = cell_dict.pop('conv')
>>> x = Tensor(np.ones([1, 10, 6, 5]), ms.float32)
>>> x = pop_cell(x)
>>> print(x.shape)
(1, 6, 2, 1)
```

(continues on next page)

(continued from previous page)

```
>>> print(len(cell_dict))
2
```

clear()
Remove all Cells from the CellDict.

items()
Return an iterable of the CellDict key-value pairs.

Returns

An iterable object.

keys()
Return an iterable of the CellDict keys.

Returns

An iterable object.

pop(*key*)
Remove key from the CellDict and return its cell.

Parameters

key (*str*) – key to pop from the CellDict.

Raises

KeyError – If *key* not exist in CellDict when attempt to access cell.

update(*cells*)
Update the CellDict by overwriting the existing keys with the key-value pairs from a mapping or an iterable.

Parameters

cells (*iterable*) – An iterable of key-value pairs of (key, Cell), the type of key-value pairs is (string, Cell); Or a mapping(dictionary) from string to Cell. The type of Cell can not be CellDict, CellList or SequentialCell. The key can not be same with the attributes of class Cell, can not contain '.', can not be an empty string.

Note: If the *cells* is a CellDict, an OrderedDict or an iterable containing key-value pairs, the order of newly added elements is maintained.

Raises

- **TypeError** – If *cells* is not an iterable object.
- **TypeError** – If key-value pairs in *cells* are not iterable objects.
- **ValueError** – If the length of key-value pairs in *cells* is not 2.
- **TypeError** – If the cell in *cells* is None.
- **TypeError** – If the type of cell in *cells* is not Cell.
- **TypeError** – If the type of cell in *cells* is CellDict, CellList or SequentialCell.
- **TypeError** – If the type of key in *cells* is not string.
- **KeyError** – If the key in *cells* is same with the attributes of class Cell.
- **KeyError** – If the key in *cells* contain ".".

- **KeyError** –If the key in *cells* is an empty string.

values()
Return an iterable of the CellDict values.

Returns

An iterable object.

3.2.2 mindspore.nn.CellList

class mindspore.nn.**CellList**(**args*, ***kwargs*)
Holds Cells in a list. For more details about Cell, please refer to Cell.

CellList can be used like a regular Python list, the Cells it contains have been initialized and the types of Cells it contains can not be CellDict. Unlike the SequentialCell, the cells in CellList are not connected.

Parameters

args (*list, optional*) –List of subclass of Cell.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>>
>>> conv = ms.nn.Conv2d(100, 20, 3)
>>> bn = ms.nn.BatchNorm2d(20)
>>> relu = ms.nn.ReLU()
>>> cell_ls = ms.nn.CellList([bn])
>>> cell_ls.insert(0, conv)
>>> cell_ls.append(relu)
>>> cell_ls.extend([relu, relu])
>>> cell_ls_3 = cell_ls[3]
>>> input1 = ms.Tensor(np.ones([2, 3]), ms.float32)
>>> output = cell_ls_3(input1)
>>> print(output)
[[1. 1. 1.]
[1. 1. 1.]]
```

append(*cell*)
Appends a given Cell to the end of the list.

Parameters

cell (Cell) –The subcell to be appended.

extend(*cells*)
Appends Cells from a Python iterable to the end of the list.

Parameters

cells (*list*) –The Cells to be extended, the types of Cells can not be CellDict.

Raises

TypeError –If the argument cells are not a list of Cells.

insert (*index*, *cell*)
Inserts a given Cell before a given index in the list.

Parameters

- **index** (*int*) –The Insert index in the CellList.
- **cell** (Cell) –The Cell to be inserted.

3.2.3 mindspore.nn.SequentialCell

class mindspore.nn.**SequentialCell**(**args*)
Sequential Cell container. For more details about Cell, please refer to Cell.

A list of Cells will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of cells can also be passed in.

Parameters

args (*list, OrderedDict*) –List or OrderedDict of subclass of Cell.

Inputs:

- **x** (Tensor) - Tensor with shape according to the first Cell in the sequence.

Outputs:

Tensor, the output Tensor with shape depending on the input *x* and defined sequence of Cells.

Raises

TypeError –If the type of the *args* is not list or OrderedDict.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>>
>>> conv = nn.Conv2d(3, 2, 3, pad_mode='valid', weight_init="ones")
>>> relu = nn.ReLU()
>>> seq = nn.SequentialCell([conv, relu])
>>> x = Tensor(np.ones([1, 3, 4, 4]), dtype = mindspore.float32)
>>> output = seq(x)
>>> print(output)
[[[[27. 27.]
   [27. 27.]]
  [[27. 27.]
   [27. 27.]]]]
>>> from collections import OrderedDict
>>> d = OrderedDict()
>>> d["conv"] = conv
```

(continues on next page)

(continued from previous page)

```
>>> d["relu"] = relu
>>> seq = nn.SequentialCell(d)
>>> x = Tensor(np.ones([1, 3, 4, 4]), dtype=mindspore.float32)
>>> output = seq(x)
>>> print(output)
[[[[27. 27.]
   [27. 27.]]
  [[27. 27.]
   [27. 27.]]]]
```

append(*cell*)

Appends a given Cell to the end of the list.

Parameters

cell (Cell) –The Cell to be appended.

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>>
>>> conv = nn.Conv2d(3, 2, 3, pad_mode='valid', weight_init="ones")
>>> bn = nn.BatchNorm2d(2)
>>> relu = nn.ReLU()
>>> seq = nn.SequentialCell([conv, bn])
>>> seq.append(relu)
>>> x = Tensor(np.ones([1, 3, 4, 4]), dtype=mindspore.float32)
>>> output = seq(x)
>>> print(output)
[[[[26.999863 26.999863]
   [26.999863 26.999863]]
  [[26.999863 26.999863]
   [26.999863 26.999863]]]]
```

3.3 Wrapper Layer

API Name	Description	Supported Platforms
mindspore.nn.DistributedGradReducer	A distributed optimizer.	Ascend GPU
mindspore.nn.DynamicLossScaleUpdateCell	Dynamic Loss scale update cell.	Ascend GPU
mindspore.nn.FixedLossScaleUpdateCell	Update cell with fixed loss scaling value.	Ascend GPU
mindspore.nn.ForwardValueAndGrad	Encapsulate training network.	Ascend GPU CPU
mindspore.nn.GetNextSingleOp	Cell to run for getting the next operation.	Ascend GPU
mindspore.nn.GradAccumulationCell	Wrap the network with Micro Batch to enable the grad accumulation in semi_auto_parallel/auto_parallel mode.	Ascend GPU
mindspore.nn.ParameterUpdate	Cell that updates parameter.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.nn.PipelineCell	Slice MiniBatch into finer-grained MicroBatch for use in pipeline-parallel training.	Ascend GPU
mindspore.nn.TimeDistributed	The time distributed layer.	Ascend GPU CPU
mindspore.nn.TrainOneStepCell	Network training package class.	Ascend GPU CPU
mindspore.nn.TrainOneStepWithLossScaleCell	Network training with loss scaling.	Ascend GPU
mindspore.nn.WithEvalCell	Wraps the forward network with the loss function.	Ascend GPU CPU
mindspore.nn.WithLossCell	Cell with loss function.	Ascend GPU CPU

3.3.1 mindspore.nn.DistributedGradReducer

class mindspore.nn.**DistributedGradReducer**(*parameters*, *mean=None*, *degree=None*, *fusion_type=1*, *group=GlobalComm.WORLD_COMM_GROUP*)

A distributed optimizer.

Aggregate the gradients for all cards by using AllReduce in data parallel.

Parameters

- **parameters** (*list*) –the parameters to be updated.
- **mean** (*bool*) –When mean is true, the mean coefficient (degree) would apply on gradients. When it is not specified, using the configuration *gradients_mean* in auto_parallel_context. Default: `None` .
- **degree** (*int*) –The mean coefficient. Usually it equals to device number. Default: `None` .
- **fusion_type** (*int*) –The type of all reduce fusion. Default: `1` .
- **group** (*str*) –The communication group to work on. Normally, the group should be created by create_group, otherwise, using the default group. Default: `GlobalComm.WORLD_COMM_GROUP` .

Raises

ValueError –If degree is not an int or less than 0.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Please see the rank table Startup for more details.

For the GPU devices, users need to prepare the host file and mpi, please see the mpirun Startup .

For the CPU device, users need to write a dynamic cluster startup script, please see the Dynamic Cluster Startup .

This example should be run with multiple devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
```

(continues on next page)

(continued from previous page)

```
>>> from mindspore import Parameter, Tensor, ops, nn
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> ms.reset_auto_parallel_context()
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.DATA_PARALLEL)
>>>
>>> class TrainingWrapper(nn.Cell):
...     def __init__(self, network, optimizer, sens=1.0):
...         super(TrainingWrapper, self).__init__(auto_prefix=False)
...         self.network = network
...         self.network.add_flags(defer_inline=True)
...         self.weights = optimizer.parameters
...         self.optimizer = optimizer
...         self.grad = ops.GradOperation(get_by_list=True, sens_param=True)
...         self.sens = sens
...         self.reducer_flag = False
...         self.grad_reducer = None
...         self.parallel_mode = context.get_auto_parallel_context("parallel_mode")
...         self.depend = ops.Depend()
...         if self.parallel_mode in [ms.ParallelMode.DATA_PARALLEL, ms.ParallelMode.HYBRID_
↪PARALLEL]:
...             self.reducer_flag = True
...         if self.reducer_flag:
...             mean = context.get_auto_parallel_context("gradients_mean")
...             degree = context.get_auto_parallel_context("device_num")
...             self.grad_reducer = nn.DistributedGradReducer(optimizer.parameters, mean,␣
↪degree)
...
...     def construct(self, *args):
...         weights = self.weights
...         loss = self.network(*args)
...         sens = F.fill(ops.DType()(loss), ops.Shape()(loss), self.sens)
...         grads = self.grad(self.network, weights)(*args, sens)
...         if self.reducer_flag:
...             # apply grad reducer on grads
...             grads = self.grad_reducer(grads)
...         return self.depend(loss, self.optimizer(grads))
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
>>>
>>> size, in_features, out_features = 16, 16, 10
>>> network = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> net_with_loss = nn.WithLossCell(network, loss)
>>> optimizer = nn.Momentum(net_with_loss.trainable_params(), learning_rate=0.1, momentum=0.
↪9)
```

(continues on next page)

(continued from previous page)

```
>>> train_cell = TrainingWrapper(net_with_loss, optimizer)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.zeros([size, out_features]).astype(np.float32))
>>> grads = train_cell(inputs, label)
>>> print(grads)
256.0
```

3.3.2 mindspore.nn.DynamicLossScaleUpdateCell

class mindspore.nn.**DynamicLossScaleUpdateCell**(*loss_scale_value*, *scale_factor*, *scale_window*)

Dynamic Loss scale update cell.

For loss scaling training, the initial loss scaling value will be set to be *loss_scale_value*. In each training step, the loss scaling value will be decreased by $loss\_scale/scale\_factor$ when there is an overflow. And it will be increased by $loss\_scale*scale\_factor$ if there is no overflow for a continuous *scale_window* steps.

get_update_cell method of *mindspore.amp.DynamicLossScaleManager* will return this class. It will be called by *mindspore.nn.TrainOneStepWithLossScaleCell* during training to update loss scale.

Parameters

- **loss_scale_value** (*float*) –Initializes loss scale.
- **scale_factor** (*int*) –Coefficient of increase and decrease.
- **scale_window** (*int*) –Maximum continuous training steps that do not have overflow to increase the loss scale.

Inputs:

- **loss_scale** (Tensor) - The loss scale value during training with shape ().
- **overflow** (bool) - Whether the overflow occurs or not.

Outputs:

bool, the input *overflow*.

Supported Platforms:

Ascend GPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, Parameter, nn, ops
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
```

(continues on next page)

(continued from previous page)

```
...         return output
...
>>> in_features, out_features = 16, 10
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = nn.WithLossCell(net, loss)
>>> manager = nn.DynamicLossScaleUpdateCell(loss_scale_value=2**12, scale_factor=2, scale_
↪window=1000)
>>> train_network = nn.TrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
↪sense=manager)
>>> input = Tensor(np.ones([out_features, in_features]), mindspore.float32)
>>> labels = Tensor(np.ones([out_features,]), mindspore.float32)
>>> output = train_network(input, labels)
```

get_loss_scale()

Get Loss Scale value.

Returns

float, the loss scale value.

Examples

```
>>> from mindspore import nn
>>> manager = nn.DynamicLossScaleUpdateCell(loss_scale_value=212, scale_factor=2, scale_
↪window=1000)
>>> output = manager.get_loss_scale()
>>> print(output)
212
```

3.3.3 mindspore.nn.FixedLossScaleUpdateCell

class mindspore.nn.**FixedLossScaleUpdateCell**(*loss_scale_value*)

Update cell with fixed loss scaling value.

get_update_cell method of *mindspore.amp.FixedLossScaleManager* will return this class. It will be called by *mindspore.nn.TrainOneStepWithLossScaleCell* during trainning.

Parameters

loss_scale_value (*float*) –Initializes loss scale.

Inputs:

- **loss_scale** (Tensor) - The loss scale value during training with shape (), it is ignored in this class.
- **overflow** (bool) - Whether the overflow occurs or not.

Outputs:

bool, the input *overflow*.

Supported Platforms:

Ascend GPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, Parameter, nn, ops
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
...
>>> in_features, out_features = 16, 10
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = nn.WithLossCell(net, loss)
>>> manager = nn.FixedLossScaleUpdateCell(loss_scale_value=2**12)
>>> train_network = nn.TrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
↪sense=manager)
>>> input = Tensor(np.ones([out_features, in_features]), mindspore.float32)
>>> labels = Tensor(np.ones([out_features,]), mindspore.float32)
>>> output = train_network(input, labels)
```

get_loss_scale()

Get Loss Scale value.

Returns

float, the loss scale value.

Examples

```
>>> from mindspore import nn
>>> manager = nn.FixedLossScaleUpdateCell(loss_scale_value=212)
>>> output = manager.get_loss_scale()
>>> print(output)
212
```

3.3.4 mindspore.nn.ForwardValueAndGrad

class mindspore.nn.**ForwardValueAndGrad**(*network*, *weights=None*, *get_all=False*, *get_by_list=False*, *sens_param=False*)

Encapsulate training network.

Including the network and a gradient function. The resulting Cell is trained with input '*inputs'. The backward graph will be created in the gradient function to calculating gradient.

Parameters

- **network** (*Union[Cell, Function, MethodType]*) –The training network.
- **weights** (ParameterTuple) –The parameters of the training network that need to calculate the gradient. Default: None .
- **get_all** (*bool*) –If True , get all the gradients with respect to inputs. Default: False .
- **get_by_list** (*bool*) –If True s, get all the gradients with respect to Parameter variables. If get_all and get_by_list are both False , get the gradient with respect to first input. If get_all and get_by_list are both True , get the gradients with respect to inputs and Parameter variables at the same time in the form of ((gradients with respect to inputs), (gradients with respect to parameters)). Default: False .
- **sens_param** (*bool*) –Whether to append sensitivity (gradient with respect to output) as input. If sens_param is False , a 'ones_like(outputs)' sensitivity will be attached automatically. Default: False . If the sens_param is True , a sensitivity (gradient with respect to output) needs to be transferred through the input parameter.

Inputs:

- ***inputs** (Tuple(Tensor···)) - Tuple of inputs with shape $(N, \ldots)$.
- **sens** - A sensitivity (gradient with respect to output) as the input of backpropagation. If network has single output, the sens is a tensor. If network has multiple outputs, the sens is the tuple(tensor).

Outputs:

- **forward value** - The result of network forward running.
- **gradients** (tuple(tensor)) - The gradients of network parameters and inputs.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, nn, ops, ParameterTuple, Parameter
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="weight")
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         out = self.matmul(x, self.weight)
...         return out
...
>>> net = Net()
>>> criterion = nn.SoftmaxCrossEntropyWithLogits()
>>> net_with_criterion = nn.WithLossCell(net, criterion)
>>> weight = ParameterTuple(net.trainable_params())
>>> train_network = nn.ForwardValueAndGrad(net_with_criterion, weights=weight, get_all=True,␣
↪get_by_list=True)
>>> inputs = Tensor(np.ones([1, 2]).astype(np.float32))
>>> labels = Tensor(np.ones([1, 2]).astype(np.float32))
>>> result = train_network(inputs, labels)
>>> print(result)
 (Tensor(shape=[1], dtype=Float32, value= [ 1.38629436e+00]), ((Tensor(shape=[1, 2],␣
↪dtype=Float32, value=
```

(continues on next page)

(continued from previous page)

```
[[ -1.00000000e+00,  -1.00000000e+00]]), Tensor(shape=[1, 2], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00]])), (Tensor(shape=[2, 2], dtype=Float32, value=
[[ -5.00000000e-01,  -5.00000000e-01],
 [ -5.00000000e-01,  -5.00000000e-01]]),)))
```

3.3.5 mindspore.nn.GetNextSingleOp

class mindspore.nn.**GetNextSingleOp**(*dataset_types*, *dataset_shapes*, *queue_name*)

Cell to run for getting the next operation.

For detailed information, refer to *mindspore.ops.GetNext*.

Parameters

- **dataset_types** (list[*mindspore.dtype*]) –The types of dataset.
- **dataset_shapes** (*list[tuple[int]]*) –The shapes of dataset.
- **queue_name** (*str*) –Queue name to fetch the data.

Outputs:

tuple[Tensor], the data gets from Dataset.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import ops, nn
>>> from mindspore import dataset as ds
>>> from mindspore import dtype as mstype
>>>
>>> data_path =  "/path/to/MNIST_Data/train/"
>>> train_dataset = ds.MnistDataset(data_path, num_samples=10)
>>> dataset_helper = mindspore.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>> dataset = dataset_helper.iter.dataset
>>> dataset_types, dataset_shapes = dataset_helper.types_shapes()
>>> queue_name = dataset.__transfer_dataset__.queue_name
>>> get_next_single_op_net = nn.GetNextSingleOp(dataset_types, dataset_shapes, queue_name)
>>> data, label = get_next_single_op_net()
>>> relu = ops.ReLU()
>>> result = relu(data.astype(mstype.float32))
>>> print(result.shape)
(28, 28, 1)
```

3.3.6 mindspore.nn.GradAccumulationCell

class mindspore.nn.**GradAccumulationCell**(*network*, *micro_size*)

Wrap the network with Micro Batch to enable the grad accumulation in semi_auto_parallel/auto_parallel mode.

Note: The api will be deprecated, please use the api *mindspore.parallel.nn.GradAccumulation* instead.

Parameters

- **network** (Cell) –The target network to wrap.
- **micro_size** (*int*) –MicroBatch size.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = nn.GradAccumulationCell(net, 4)
```

3.3.7 mindspore.nn.ParameterUpdate

class mindspore.nn.**ParameterUpdate**(*param*)

Cell that updates parameter.

With this Cell, one can manually update *param* with the input *Tensor*.

Parameters

param (Parameter) –The parameter to be updated manually.

Inputs:

- **x** (Tensor) - A tensor whose shape and type are the same with *param*.

Outputs:

Tensor, the updated value.

Raises

KeyError –If parameter with the specified name does not exist.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, Tensor
>>> network = nn.Dense(3, 4)
>>> param = network.parameters_dict()['weight']
>>> update = nn.ParameterUpdate(param)
>>> update.phase = "update_param"
>>> weight = Tensor(np.arange(12).reshape((4, 3)), mindspore.float32)
>>> output = update(weight)
>>> print(output)
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]
 [ 9. 10. 11.]]
```

3.3.8 mindspore.nn.PipelineCell

class mindspore.nn.**PipelineCell**(*network*, *micro_size*, *stage_config=None*)

Slice MiniBatch into finer-grained MicroBatch for use in pipeline-parallel training.

Note:

- micro_size must be greater or equal to pipeline stages.
- The api will be deprecated, please use the api *mindspore.parallel.nn.Pipeline* instead.

Parameters

- **network** (Cell) – The target network to wrap.
- **micro_size** (*int*) – MicroBatch size.
- **stage_config** (*dict, optional*) – The stage configuration for each cell's execution in pipeline parallel. Default None.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = nn.PipelineCell(net, 4)
```

3.3.9 mindspore.nn.TimeDistributed

class mindspore.nn.**TimeDistributed**(*layer*, *time_axis*, *reshape_with_axis=None*)

The time distributed layer.

Time distributed is a wrapper which allows to apply a layer to every temporal slice of an input. And the *x* should be at least 3D. There are two cases in the implementation. When reshape_with_axis provided, the reshape method will be chosen, which is more efficient; otherwise, the method of dividing the inputs along time axis will be used, which is more general. For example, reshape_with_axis could not be provided when deal with Batch Normalization.

Parameters

- **layer** (*Union[Cell, Primitive]*) –The Cell or Primitive which will be wrapped.
- **time_axis** (*int*) –The axis of time_step.
- **reshape_with_axis** (*int*) –The axis which will be reshaped with time_axis. Default: None .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, T, *)$, where $*$ means any number of additional dimensions.

Outputs:

Tensor of shape $(N, T, *)$.

Raises

TypeError –If layer is not a Cell or Primitive.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.random.random([32, 10, 3]), ms.float32)
>>> dense = ms.nn.Dense(3, 6)
>>> net = ms.nn.TimeDistributed(dense, time_axis=1, reshape_with_axis=0)
>>> output = net(x)
>>> print(output.shape)
(32, 10, 6)
```

3.3.10 mindspore.nn.TrainOneStepCell

class mindspore.nn.**TrainOneStepCell**(*network*, *optimizer*, *sens=None*, *return_grad=False*)

Network training package class.

Wraps the *network* with the *optimizer*. The resulting Cell is trained with input '*inputs'. The backward graph will be created in the construct function to update the parameter. Different parallel modes are available for training.

Parameters

- **network** (Cell) –The training network. The network only supports single output.
- **optimizer** (*Union[Cell]*) –Optimizer for updating the network parameters.

- **sens** (*numbers.Number, optional*) – The scaling number to be filled as the input of backpropagation. Default value is `None` , which is `1.0` .
- **return_grad** (*bool, optional*) – Whether to return gradient. If `True`, it will return the gradient in the form of a dict while returning loss. The key of the dict is the parameter name corresponding to the gradient, and value is the gradient value. Default value is `False` .

Inputs:

- ***inputs** (Tuple(Tensor)) - Tuple of input tensors with shape $(N, \ldots)$.

Outputs:

Tensor, a tensor means the loss value, the shape of which is usually ().

Raises

TypeError – If *sens* is not a numbers.Number.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits()
>>> optim = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> #1) Using the WithLossCell provided by MindSpore
>>> loss_net = nn.WithLossCell(net, loss_fn)
>>> train_net = nn.TrainOneStepCell(loss_net, optim)
>>>
>>> #2) Using user-defined WithLossCell
>>> class MyWithLossCell(nn.Cell):
...     def __init__(self, backbone, loss_fn):
...         super(MyWithLossCell, self).__init__(auto_prefix=False)
...         self._backbone = backbone
...         self._loss_fn = loss_fn
...
...     def construct(self, x, y, label):
...         out = self._backbone(x, y)
...         return self._loss_fn(out, label)
...
...     @property
...     def backbone_network(self):
...         return self._backbone
...
>>> loss_net = MyWithLossCell(net, loss_fn)
>>> train_net = nn.TrainOneStepCell(loss_net, optim)
```

3.3.11 mindspore.nn.TrainOneStepWithLossScaleCell

class mindspore.nn.**TrainOneStepWithLossScaleCell**(*network*, *optimizer*, *scale_sense*)

Network training with loss scaling.

This is a training step with loss scaling. It takes a network, an optimizer and a scale update Cell(or a Tensor) as args. The loss scale value can be updated in both host side or device side. If you want to update it on host side, using a value of Tensor type as *scale_sense*, otherwise, using a Cell instance for updating loss scale as *scale_sense*.

Parameters

- **network** (Cell) –The training network. The network only supports single output.
- **optimizer** (Cell) –Optimizer for updating the network parameters.
- **scale_sense** (*Union[Tensor, Cell]*) –If this value is a Cell, it will be called by *TrainOneStepWithLossScaleCell* to update loss scale. If this value is a Tensor, the loss scale can be modified by *set_sense_scale*, the shape should be () or (1,).

Inputs:

- ***inputs** (Tuple(Tensor)) - Tuple of input tensors with shape $(N, \ldots)$.

Outputs:

Tuple of 3 Tensor, the loss, overflow flag and current loss scale value.

- **loss** (Tensor) - A scalar, the loss value.
- **overflow** (Tensor) - A scalar, whether overflow occur or not, the type is bool.
- **loss scale** (Tensor) - The loss scale value, the shape is () or (1,).

Raises

- **TypeError** –If *scale_sense* is neither Cell nor Tensor.
- **ValueError** –If shape of *scale_sense* is neither (1,) nor ().

Supported Platforms:

Ascend GPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, Parameter, nn, ops
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
```

(continues on next page)

(continued from previous page)

```
...
>>> size, in_features, out_features = 16, 16, 10
>>> #1) when the type of scale_sense is Cell:
>>> net = Net(in_features, out_features)
>>> loss_fn = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = nn.WithLossCell(net, loss_fn)
>>> input = Tensor(np.ones([out_features, in_features]), mindspore.float32)
>>> labels = Tensor(np.ones([out_features,]), mindspore.float32)
>>> loss = net_with_loss(input, labels)
>>> manager = nn.DynamicLossScaleUpdateCell(loss_scale_value=2**12, scale_factor=2, scale_
↪window=1000)
>>> train_network = nn.TrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
↪sense=manager)
>>> status = Tensor([0] * 8, mindspore.int32)
>>> scaling_sens = train_network.scale_sense
>>> scaling_sens_filled = ops.ones_like(loss) * ops.cast(scaling_sens, ops.dtype(loss))
>>> grads = train_network.grad(train_network.network, train_network.weights)(input, labels,␣
↪scaling_sens_filled)
>>> grads = train_network.grad_reducer(grads)
>>> cond = train_network.get_overflow_status(status, grads)
>>> overflow = train_network.process_loss_scale(cond)
>>>
>>> #2) when the type of scale_sense is Tensor:
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = nn.WithLossCell(net, loss)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.zeros([size, out_features]).astype(np.float32))
>>> scaling_sens = Tensor([1024], dtype=mindspore.float32)
>>> train_network = nn.TrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
↪sense=scaling_sens)
>>> scaling_sens = Tensor([1], dtype=mstype.float32)
>>> train_network.set_sense_scale(scaling_sens)
>>> output = train_network(inputs, label)
>>>
>>> # update scaling sens and train the network
>>> scaling_sens = Tensor([1], dtype=mindspore.float32)
>>> train_network.set_sense_scale(scaling_sens)
>>> output = train_network(inputs, label)
```

get_overflow_status(*status*, *compute_output*)

Get floating-point overflow status.

Get overflow results after executing the target process for overflow detection. User-defined training network based on this class can also call this interface to process the overflow.

Parameters

- **status** (*object*) –To control the execution sequence with start_overflow_check, it should be set as the first output of start_overflow_check.
- **compute_output** –Overflow detection should be performed in a certain computation process. Set *compute_output* as the output of the computation process.

Returns

bool, whether the overflow occurs or not.

process_loss_scale(*overflow*)
Calculate loss scale according to the overflow.

User-defined training network based on this class can also call this interface to process the overflow.

Parameters

overflow (*bool*) −Whether the overflow occurs or not.

Returns

bool, the input overflow value.

set_sense_scale(*sens*)
If the user has set the *scale_sense* of Tensor type, he can call this function to reassign the value.

Parameters

sens (Tensor) −The new sense whose shape and type are the same with original *scale_sense*.

start_overflow_check(*pre_cond*, *compute_input*)
Start floating-point overflow detection. Create and clear the overflow detection state.

Specify the argument 'pre_cond' and 'compute_input' to make sure overflow status is cleared at the right time. Taking this situation as an example, we need to execute state clearing after loss calculation and then detect overflow in the process of gradient calculation. In this case, pre_cond should be the output of the loss function, and compute_input should be the input of gradients-computing function. User-defined training network based on this class can also call this interface to process the overflow.

Parameters

- **pre_cond** (Tensor) −A precondition for starting overflow detection. It determines the executing order of overflow state clearing and prior processions. It makes sure that the function 'start_overflow' clears status after finishing the process of precondition.
- **compute_input** (*object*) −The input of subsequent process. Overflow detection should be performed on a certain computation. Set *compute_input* as the input of the computation, to ensure overflow status is cleared before executing the computation.

Returns

Tuple[object, object], the first output is used to control the execution sequence. To ensure that the *start_overflow_check* is executed before get_overflow_status after compilation optimization is performed. This value should be used as the first input of get_overflow_status. The second output is the same as the input of compute_input, used to control the execution sequence, and make ensure that the overflow flag is cleaned up when the function returns.

3.3.12 mindspore.nn.WithEvalCell

class mindspore.nn.**WithEvalCell**(*network*, *loss_fn*, *add_cast_fp32=False*)
Wraps the forward network with the loss function.

It returns loss, forward output and label to calculate the metrics.

Parameters

- **network** (Cell) −The forward network.
- **loss_fn** (Cell) −The loss function.
- **add_cast_fp32** (*bool*) −Whether to adjust the data type to float32. Default: False .

Inputs:

- **data** (Tensor) - Tensor of shape $(N, \ldots)$.
- **label** (Tensor) - Tensor of shape $(N, \ldots)$.

Outputs:

Tuple(Tensor), containing a scalar loss Tensor, a network output Tensor of shape $(N, \ldots)$ and a label Tensor of shape $(N, \ldots)$.

Raises

TypeError – If *add_cast_fp32* is not a bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>> # Define a forward network without loss function, taking LeNet5 as an example.
>>> # Refer to https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits()
>>> eval_net = nn.WithEvalCell(net, loss_fn)
```

3.3.13 mindspore.nn.WithLossCell

class mindspore.nn.**WithLossCell**(*backbone*, *loss_fn*)

Cell with loss function.

Wraps the network with loss function. This Cell accepts data and label as inputs and the computed loss will be returned.

Parameters

- **backbone** (`Cell`) – The backbone network to wrap.
- **loss_fn** (`Cell`) – The loss function used to compute loss.

Inputs:

- **data** (Tensor) - Tensor of shape $(N, \ldots)$. The dtype of *data* must be float16 or float32.
- **label** (Tensor) - Tensor of shape $(N, \ldots)$. The dtype of *label* must be float16 or float32.

Outputs:

Tensor, a tensor means the loss value, the shape of which is usually ().

Raises

TypeError – If dtype of *data* or *label* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=False)
>>> net_with_criterion = nn.WithLossCell(net, loss_fn)
>>>
>>> batch_size = 2
>>> data = Tensor(np.ones([batch_size, 1, 32, 32]).astype(np.float32) * 0.01)
>>> label = Tensor(np.ones([batch_size, 10]).astype(np.float32))
>>>
>>> output_data = net_with_criterion(data, label)
```

property backbone_network

Get the backbone network.

Returns

Cell, the backbone network.

Examples

```
>>> from mindspore import nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=False)
>>> net_with_criterion = nn.WithLossCell(net, loss_fn)
>>> backbone = net_with_criterion.backbone_network
```

3.4 Convolutional Layer

API Name	Description	Supported Platforms
mindspore.nn.Conv1d	1D convolution layer.	Ascend GPU CPU
mindspore.nn.Conv1dTranspose	Calculates a 1D transposed convolution, which can be regarded as Conv1d for the gradient of the input, also called deconvolution (although it is not an actual deconvolution).	Ascend GPU CPU
mindspore.nn.Conv2d	2D convolution layer.	Ascend GPU CPU
mindspore.nn.Conv2dTranspose	Calculates a 2D transposed convolution, which can be regarded as Conv2d for the gradient of the input, also called deconvolution (although it is not an actual deconvolution).	Ascend GPU CPU
mindspore.nn.Conv3d	3D convolution layer.	Ascend GPU CPU
mindspore.nn.Conv3dTranspose	Calculates a 3D transposed convolution, which can be regarded as Conv3d for the gradient of the input.	Ascend GPU CPU
mindspore.nn.Unfold	Extracts patches from images.	Ascend GPU

3.4.1 mindspore.nn.Conv1d

class mindspore.nn.**Conv1d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *dilation=1*, *group=1*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *dtype=mstype.float32*)

1D convolution layer.

Applies a 1D convolution over an input tensor which is typically of shape (N, C_{in}, L_{in}), where N is batch size, C is channel number, L is input sequence width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size), where kernel_size is the width of the kernel. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size})$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition and ConvNets .

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *group>1*, condition *in_channels* = *out_channels* = *group* must be satisfied.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv1d layer.
- **out_channels** (*int*) –The channel number of the output tensor of the Conv1d layer.
- **kernel_size** (*int*) –Specifies the width of the 1D convolution kernel.
- **stride** (*int, optional*) –The movement stride of the 1D convolution kernel. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input at the begin and end so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it is uniformly distributed around the input, if it is odd, the excess padding is goes to the right side. If this mode is set, *padding* must be 0.

  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible length. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding at the begin and end is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.

- **padding** (*Union(int, tuple[int], list[int]), optional*) – Specifies the amount of padding to apply on both side of *input* when *pad_mode* is set to `"pad"`. The paddings of left and right are the same, equal to padding or padding[0] when padding is a tuple of 1 integer. Default: `0` .
- **dilation** (*Union(int, tuple[int]), optional*) – Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 1 integer. Assuming $dilation = (d0,)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the width direction. The value should be in the ranges [1, L]. Default: `1` .
- **group** (*int, optional*) – Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **has_bias** (*bool, optional*) – Whether the Conv1d layer has a bias parameter. Default: `False` .
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant 'One' and 'Zero' distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'` , `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer, for more details. Default: `None` , weight will be initialized using `'HeUniform'`.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer, for more details. Default: `None` , bias will be initialized using `'Uniform'`.
- **dtype** (*mindspore.dtype*) – Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C_{in}, L_{in}) .

Outputs:

Tensor of shape (N, C_{out}, L_{out}).

pad_mode is `'same'`:

$$L_{out} = \left\lceil \frac{L_{in}}{\text{stride}} \right\rceil$$

pad_mode is `'valid'`:

$$L_{out} = \left\lfloor \frac{L_{in} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} \right\rfloor + 1$$

pad_mode is `'pad'`:

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times padding - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} \right\rfloor + 1$$

Raises

- **TypeError** – If *in_channels*, *out_channels*, *kernel_size*, *stride*, *padding* or *dilation* is not an int.
- **ValueError** – If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.

- **ValueError** –If *padding* is less than 0.
- **ValueError** –If *pad_mode* is not one of 'same', 'valid', 'pad'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Conv1d(120, 240, 4, has_bias=False, weight_init='normal')
>>> x = Tensor(np.ones([1, 120, 640]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 240, 640)
```

3.4.2 mindspore.nn.Conv1dTranspose

class mindspore.nn.**Conv1dTranspose**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *dilation=1*, *group=1*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *dtype=mstype.float32*)

Calculates a 1D transposed convolution, which can be regarded as Conv1d for the gradient of the input, also called deconvolution (although it is not an actual deconvolution).

The input is typically of shape (N, C_{in}, L_{in}), where N is batch size, C_{in} is a number of channels and L_{in} is a length of sequence.

When Conv1d and ConvTranspose1d are initialized with the same parameters, and *pad_mode* is set to 'pad', $dilation * (kernel\_size - 1) - padding$ amount of zero will be paded to both sizes of input, they are inverses of each other in regard to the input and output shapes in this case. However, when *stride* > 1, Conv1d maps multiple input shapes to the same output shape. Deconvolutional network can refer to Deconvolutional Networks.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv1dTranspose layer.
- **out_channels** (*int*) –The channel number of the output tensor of the Conv1dTranspose layer.
- **kernel_size** (*int*) –Specifies the width of the 1D convolution kernel.
- **stride** (*int*) –The movement stride of the 1D convolution kernel. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input at the begin and end so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it is uniformly distributed around the input, if it is odd, the excess padding is goes to the right side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible length. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding at the begin and end is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.

- **padding** (*int*) −The number of padding on both sides of input. The value should be greater than or equal to 0. Default: `0` .
- **dilation** (*int*) −Dilation size of 1D convolution kernel. If $k > 1$, the kernel is sampled every k elements. The value of k is in range of [1, L]. Default: `1` .
- **group** (*int*) −Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. When *group* > 1, the Ascend platform is not supported yet. Default: `1` .
- **has_bias** (*bool*) −Whether the Conv1dTranspose layer has a bias parameter. Default: `False`.
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number]*) −Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant `'One'` and `'Zero'` distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'`, `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer for more details. Default: `None` , weight will be initialized using HeUniform.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number]*) −Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer for more details. Default: `None` , bias will be initialized using Uniform.
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C_{in}, L_{in}).

Outputs:

Tensor of shape (N, C_{out}, L_{out}).

pad_mode is `'same'`: $L_{out} = \frac{L_{in}+\text{stride}-1}{\text{stride}}$

pad_mode is `'valid'`: $L_{out} = (L_{in} - 1) \times \text{stride} + \text{dilation} \times (\text{kernel\_size} - 1) + 1$

pad_mode is `'pad'`: $L_{out} = (L_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{dilation} \times (\text{kernel\_size} - 1) + 1$

Raises

- **TypeError** −If *in_channels*, *out_channels*, *kernel_size*, *stride*, *padding* or *dilation* is not an int.
- **ValueError** −If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** −If *padding* is less than 0.
- **ValueError** −If *pad_mode* is not one of 'same', 'valid', 'pad'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Conv1dTranspose(3, 64, 4, has_bias=False, weight_init='normal', pad_mode='pad')
>>> x = Tensor(np.ones([1, 3, 50]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 64, 53)
```

3.4.3 mindspore.nn.Conv2d

class mindspore.nn.**Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *dilation=1*, *group=1*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *data_format='NCHW'*, *dtype=mstype.float32*)

2D convolution layer.

Applies a 2D convolution over an input tensor which is typically of shape $(N, C_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, H is feature height, W is feature width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1]), where kernel_size[0] and kernel_size[1] are the height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *group>1*, condition *in_channels* = *out_channels* = *group* must be satisfied.

Parameters

- **in_channels** (*int*) −The channel number of the input tensor of the Conv2d layer.

- **out_channels** (*int*) –The channel number of the output tensor of the Conv2d layer.
- **kernel_size** (*Union[int, tuple[int]]*) –Specifies the height and width of the 2D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the height and width of the convolution kernel. A tuple of two integers represents the height and width of the convolution kernel respectively.
- **stride** (*Union[int, tuple[int]], optional*) –The movement stride of the 2D convolution kernel. The data type is an integer or a tuple of two or four integers. An integer represents the movement step size in both height and width directions. A tuple of two integers represents the movement step size in the height and width directions respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union[int, tuple[int]], optional*) –The number of padding on the height and width directions of the input. The data type is an integer or a tuple of four integers. If *padding* is an integer, then the top, bottom, left, and right padding are all equal to *padding*. If *padding* is a tuple of 4 integers, then the top, bottom, left, and right padding is equal to *padding[0]*, *padding[1]*, *padding[2]*, and *padding[3]* respectively. The value should be greater than or equal to 0. Default: `0` .
- **dilation** (*Union(int, tuple[int]), optional*) –Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 2 or 4 integers. A single int means the dilation size is the same in both the height and width directions. A tuple of two ints represents the dilation size in the height and width directions, respectively. For a tuple of four ints, the two ints correspond to (N, C) dimension are treated as 1, and the two correspond to (H, W) dimensions is the dilation size in the height and width directions respectively. Assuming $dilation = (d0, d1)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the height direction and $d1 - 1$ elements in the width direction. The values in the height and width dimensions are in the ranges [1, H] and [1, W], respectively. Default: `1` .
- **group** (*int, optional*) –Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. If the group is equal to *in_channels* and *out_channels*, this 2D convolution layer also can be called 2D depthwise convolution layer. Default: `1` .
- **has_bias** (*bool, optional*) –Whether the Conv2d layer has a bias parameter. Default: `False` .
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant `'One'` and `'Zero'` distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'` , `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer, for more details. Default: `None` , weight will be initialized using `'HeUniform'`.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer, for more details. Default: `None` , bias will be initialized using `'Uniform'` .
- **data_format** (*str, optional*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` . (NHWC is only supported in GPU now.)
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or $(N, H_{in}, W_{in}, C_{in})$.

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$ or $(N, H_{out}, W_{out}, C_{out})$.

pad_mode is `'same'`:

$$\begin{aligned} H_{out} &= \left\lceil \frac{H_{in}}{\text{stride}[0]} \right\rceil \\ W_{out} &= \left\lceil \frac{W_{in}}{\text{stride}[1]} \right\rceil \end{aligned}$$

pad_mode is `'valid'`:

$$\begin{aligned} H_{out} &= \left\lfloor \frac{H_{in} - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \end{aligned}$$

pad_mode is `'pad'`:

$$\begin{aligned} H_{out} &= \left\lfloor \frac{H_{in} + padding[0] + padding[1] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} + padding[2] + padding[3] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \end{aligned}$$

Raises

- **TypeError** – If *in_channels*, *out_channels* or *group* is not an int.
- **TypeError** – If *kernel_size*, *stride*, *padding* or *dilation* is neither an int not a tuple.
- **ValueError** – If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *padding* is less than 0.
- **ValueError** – If *pad_mode* is not one of 'same', 'valid', 'pad'.
- **ValueError** – If *padding* is a tuple whose length is not equal to 4.
- **ValueError** – If *pad_mode* is not equal to 'pad' and *padding* is not equal to (0, 0, 0, 0).
- **ValueError** – If *data_format* is neither 'NCHW' nor 'NHWC'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Conv2d(120, 240, 4, has_bias=False, weight_init='normal')
>>> x = Tensor(np.ones([1, 120, 1024, 640]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 240, 1024, 640)
```

3.4.4 mindspore.nn.Conv2dTranspose

class mindspore.nn.**Conv2dTranspose**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *output_padding=0*, *dilation=1*, *group=1*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *dtype=mstype.float32*)

Calculates a 2D transposed convolution, which can be regarded as Conv2d for the gradient of the input, also called deconvolution (although it is not an actual deconvolution).

The input is typically of shape $(N, C_{in}, H_{in}, W_{in})$, where N is batch size, C_{in} is space dimension, H_{in}, W_{in} are the height and width of the feature layer respectively.

When Conv2d and Conv2dTranspose are initialized with the same parameters, and *pad_mode* is set to 'pad', $dilation * (kernel\_size - 1) - padding$ amount of zero will be paded to the height and width directions of the input, they are inverses of each other in regard to the input and output shapes in this case. However, when *stride* > 1, Conv2d maps multiple input shapes to the same output shape. Deconvolutional network can refer to Deconvolutional Networks.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv2dTranspose layer.
- **out_channels** (*int*) –The channel number of the output tensor of the Conv2dTranspose layer.
- **kernel_size** (*Union[int, tuple[int]]*) –Specifies the height and width of the 2D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the height and width of the convolution kernel. A tuple of two integers represents the height and width of the convolution kernel respectively.
- **stride** (*Union[int, tuple[int]]*) –The movement stride of the 2D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the movement step size in both height and width directions. A tuple of two integers represents the movement step size in the height and width directions respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union[int, tuple[int]]*) –The number of padding on the height and width directions of the input. The data type is an integer or a tuple of four integers. If *padding* is an integer, then the top, bottom, left, and right padding are all equal to *padding*. If *padding* is a tuple of 4 integers, then the top, bottom, left, and right padding is equal to *padding[0]*, *padding[1]*, *padding[2]*, and *padding[3]* respectively. The value should be greater than or equal to 0. Default: `0` .
- **output_padding** (*Union[int, tuple[int]]*) –The number of padding on the height and width directions of the output. The data type is an integer or a tuple of two integers. If *output_padding* is an integer, then the bottom and right padding are all equal to *output_padding*. If *output_padding* is a tuple of 2 integers, then the bottom and right padding is equal to *output_padding[0]*, *output_padding[1]* respectively. If *output_padding* is not equal to 0, *pad_mode* must be *pad*. The value should be in range of *[0, max(stride, dilation))* . Default: `0` .
- **dilation** (*Union[int, tuple[int]]*) –Dilation size of 2D convolution kernel. It can be a single int or a tuple of 2 integers. A single int means the dilation size is the same in both the height and width directions. A tuple of two ints represents the dilation size in the height and width directions, respectively. Assuming $dilation = (d0, d1)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the height direction and $d1 - 1$ elements

in the width direction. The values in the height and width dimensions are in the ranges [1, H] and [1, W], respectively. Default: `1` .

- **group** (*int*) −Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **has_bias** (*bool*) −Whether the Conv2dTranspose layer has a bias parameter. Default: `False` .
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number]*) −Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant `'One'` and `'Zero'` distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'` , `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer for more details. Default: `None` , weight will be initialized using HeUniform.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number]*) −Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer for more details. Default: `None` , bias will be initialized using Uniform.
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$.

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$.

pad_mode is `'same'`:

$$H_{out} = \mathrm{H}_{in} \times \text{stride}[0]$$
$$W_{out} = \mathrm{W}_{in} \times \text{stride}[1]$$

pad_mode is `'valid'`:

$$H_{out} = \mathrm{H}_{in} \times \text{stride}[0] + \max\{(\text{dilation}[0] - 1) \times (\text{kernel\_size}[0] - 1) - \text{stride}[0], 0\}$$
$$W_{out} = \mathrm{W}_{in} \times \text{stride}[1] + \max\{(\text{dilation}[1] - 1) \times (\text{kernel\_size}[1] - 1) - \text{stride}[1], 0\}$$

pad_mode is `'pad'`:

$$H_{out} = \mathrm{H}_{in} \times \text{stride}[0] - (padding[0] + padding[1]) + \text{kernel\_size}[0] + (\text{dilation}[0] - 1) \times (\text{kernel\_size}[0] - 1) - \text{stride}[0] + \text{outp}$$
$$W_{out} = \mathrm{W}_{in} \times \text{stride}[1] - (padding[2] + padding[3]) + \text{kernel\_size}[1] + (\text{dilation}[1] - 1) \times (\text{kernel\_size}[1] - 1) - \text{stride}[1] + \text{out}$$

Raises

- **TypeError** −If *in_channels*, *out_channels* or *group* is not an int.
- **TypeError** −If *kernel_size*, *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **ValueError** −If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** −If *padding* is less than 0.
- **ValueError** −If *pad_mode* is not one of 'same', 'valid', 'pad'.
- **ValueError** −If *padding* is a tuple whose length is not equal to 4.
- **ValueError** −If *pad_mode* is not equal to 'pad' and *padding* is not equal to (0, 0, 0, 0).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Conv2dTranspose(3, 64, 4, has_bias=False, weight_init='normal', pad_mode='pad')
>>> x = Tensor(np.ones([1, 3, 16, 50]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 64, 19, 53)
```

3.4.5 mindspore.nn.Conv3d

class mindspore.nn.**Conv3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *dilation=1*, *group=1*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *data_format='NCDHW'*, *dtype=mstype.float32*)

3D convolution layer.

Applies a 3D convolution over an input tensor which is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, D, H, W are the depth, height and width of the feature map, respectively.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{out_j})$ represents the bias of the j-th output channel, $weight(C_{out_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1], kernel_size[2]) where kernel_size[0], kernel_size[1] and kernel_size[2] are the depth, height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *group>1*, condition *in_channels* = *out_channels* = *group* must be satisfied.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv3d layer.
- **out_channels** (*int*) –The channel number of the output tensor of the Conv3d layer.
- **kernel_size** (*Union[int, tuple[int]]*) –Specifies the depth, height and width of the 3D convolution kernel. It can be a single int or a tuple of 3 integers. A single int means the value is for depth, height and the width. A tuple of 3 ints means the first value is for depth and the rest is for the height and width.
- **stride** (*Union[int, tuple[int]], optional*) –The movement stride of the 3D convolution kernel. The data type is an integer or a tuple of three integers. An integer represents the movement step size in depth, height and width directions. A tuple of three integers represents the movement step size in the depth, height and width directions respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int]), optional*) –The number of padding on the depth, height and width directions of the input. The data type is an integer or a tuple of six integers. If *padding* is an integer, then the head, tail, top, bottom, left, and right padding are all equal to *padding*. If *padding* is a tuple of six integers, then the head, tail, top, bottom, left, and right padding is equal to *padding[0]*, *padding[1]*, *padding[2]*, *padding[3]*, *padding[4]* and *padding[5]* respectively. The value should be greater than or equal to 0. Default: `0` .
- **dilation** (*Union[int, tuple[int]], optional*) –Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 3 integers. A single int means the dilation size is the same in the depth, height and width directions. A tuple of 3 ints represents the dilation size in the depth, height and width directions, respectively. Assuming $dilation = (d0, d1, d2)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the depth direction, $d1 - 1$ elements in the height direction, $d2 - 1$ elements in the width direction respectively. The values in the depth, height and width dimensions are in the ranges [1, D], [1, H] and [1, W], respectively. Default: `1` .
- **group** (*int, optional*) –Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **has_bias** (*bool, optional*) –Whether the Conv3d layer has a bias parameter. Default: `False` .
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant `'One'` and `'Zero'` distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'` , `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer, for more details. Default: `None` , weight will be initialized using `'HeUniform'`.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer, for more details. Default: `None` , bias will be initialized using `'Uniform'` .
- **data_format** (*str, optional*) –The optional value for data format. Currently only support `'NCDHW'` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$. Currently, input data type support float16 and float32 in CPU and GPU, and only float16 in Ascend.

Outputs:

Tensor of shape is $(N, C_{out}, D_{out}, H_{out}, W_{out})$.

pad_mode is `'same'` :

$$\begin{aligned} D_{out} &= \left\lceil \frac{D_{in}}{\text{stride}[0]} \right\rceil \\ H_{out} &= \left\lceil \frac{H_{in}}{\text{stride}[1]} \right\rceil \\ W_{out} &= \left\lceil \frac{W_{in}}{\text{stride}[2]} \right\rceil \end{aligned}$$

pad_mode is `'valid'` :

$$\begin{aligned} D_{out} &= \left\lfloor \frac{D_{in} - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ H_{out} &= \left\lfloor \frac{H_{in} - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} \right\rfloor + 1 \end{aligned}$$

pad_mode is `'pad'` :

$$\begin{aligned} D_{out} &= \left\lfloor \frac{D_{in} + padding[0] + padding[1] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ H_{out} &= \left\lfloor \frac{H_{in} + padding[2] + padding[3] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} + padding[4] + padding[5] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} \right\rfloor + 1 \end{aligned}$$

Raises

- **TypeError** – If *in_channels*, *out_channels* or *group* is not an int.
- **TypeError** – If *kernel_size*, *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **ValueError** – If *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *padding* is less than 0.
- **ValueError** – If *pad_mode* is not one of 'same', 'valid', 'pad'.
- **ValueError** – If *padding* is a tuple whose length is not equal to 6.
- **ValueError** – If *pad_mode* is not equal to 'pad' and *padding* is not equal to (0, 0, 0, 0, 0, 0).
- **ValueError** – If *data_format* is not 'NCDHW'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.ones([16, 3, 10, 32, 32]), mindspore.float32)
>>> conv3d = nn.Conv3d(in_channels=3, out_channels=32, kernel_size=(4, 3, 3))
>>> output = conv3d(x)
>>> print(output.shape)
(16, 32, 10, 32, 32)
```

3.4.6 mindspore.nn.Conv3dTranspose

class mindspore.nn.**Conv3dTranspose**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *pad_mode='same'*, *padding=0*, *dilation=1*, *group=1*, *output_padding=0*, *has_bias=False*, *weight_init=None*, *bias_init=None*, *data_format='NCDHW'*, *dtype=mstype.float32*)

Calculates a 3D transposed convolution, which can be regarded as Conv3d for the gradient of the input. It also called deconvolution (although it is not an actual deconvolution).

The input is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C_{in} is a number of channels, D_{in}, H_{in}, W_{in} are the depth, height and width of the feature layer respectively.

When Conv3d and Conv3dTranspose are initialized with the same parameters, and *pad_mode* is set to `'pad'`, $dilation * (kernel\_size - 1) - padding$ amount of zero will be paded to the depth, height and width directions of the input, they are inverses of each other in regard to the input and output shapes in this case. However, when *stride* > 1, Conv2d maps multiple input shapes to the same output shape. For the detailed information of Deconvolutional network, refer to Deconvolutional Networks.

Note: For Atlas A2 training series products, *output_padding* is currently not supported.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv3dTranspose layer.
- **out_channels** (*int*) –The channel number of the output tensor of the Conv3dTranspose layer.
- **kernel_size** (*Union[int, tuple[int]]*) –Specifies the depth, height and width of the 3D convolution kernel. The data type is an integer or a tuple of three integers. An integer represents the depth, height and width of the convolution kernel. A tuple of three integers represents the depth, height and width of the convolution kernel respectively.
- **stride** (*Union[int, tuple[int]], optional*) –The movement stride of the 3D convolution kernel. The data type is an integer or a tuple of three integers. An integer represents the movement step size in depth, height and width directions. A tuple of three integers represents the movement step size in the depth, height and width directions respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"same"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.

  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.

- **padding** (*Union(int, tuple[int]), optional*) – The number of padding on the depth, height and width directions of the input. The data type is an integer or a tuple of six integers. If *padding* is an integer, then the head, tail, top, bottom, left, and right padding are all equal to *padding*. If *padding* is a tuple of six integers, then the head, tail, top, bottom, left, and right padding is equal to *padding[0]*, *padding[1]*, *padding[2]*, *padding[3]*, *padding[4]* and *padding[5]* respectively. The value should be greater than or equal to 0. Default: `0` .
- **dilation** (*Union[int, tuple[int]], optional*) – Specifies the dilation rate to use for dilated convolution. The data type can be a single int or a tuple of 3 integers. A single int means the dilation size is the same in the depth, height and width directions. A tuple of 3 ints represents the dilation size in the depth, height and width directions, respectively. Assuming $dilation = (d0, d1, d2)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the depth direction, $d1-1$ elements in the height direction, $d2-1$ elements in the width direction respectively. The values in the depth, height and width dimensions are in the ranges [1, D], [1, H] and [1, W], respectively. Default: `1` .
- **group** (*int, optional*) – Splits filter into groups, *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **output_padding** (*Union(int, tuple[int]), optional*) – The number of padding on the depth, height and width directions of the output. The data type is an integer or a tuple of three integers. If *output_padding* is an integer, then the depth, height, and width dimension padding are all equal to *output_padding*. If *output_padding* is a tuple of three integers, then the depth, height, and width padding is equal to *output_padding[0]*, *output_padding[1]* and *output_padding[2]* respectively. The value should be greater than or equal to 0. Default: `0` .
- **has_bias** (*bool, optional*) – Whether the Conv3dTranspose layer has a bias parameter. Default: `False` .
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initialization method of weight parameter. It can be a Tensor, a string, an Initializer or a numbers.Number. When a string is specified, values from `'TruncatedNormal'` , `'Normal'` , `'Uniform'` , `'HeUniform'` and `'XavierUniform'` distributions as well as constant `'One'` and `'Zero'` distributions are possible. Alias `'xavier_uniform'` , `'he_uniform'` , `'ones'` and `'zeros'` are acceptable. Uppercase and lowercase are both acceptable. Refer to the values of Initializer for more details. Default: `None` , weight will be initialized using HeUniform.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initialization method of bias parameter. Available initialization methods are the same as 'weight_init'. Refer to the values of Initializer for more details. Default: `None` , bias will be initialized using Uniform.
- **data_format** (*str, optional*) – The optional value for data format. Currently only support `'NCDHW'` . Default: `'NCDHW'` .
- **dtype** (*mindspore.dtype*, optional) – Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$. Currently input data dtype only supports float16 and float32.

Outputs:

Tensor, the shape is $(N, C_{out}, D_{out}, H_{out}, W_{out})$.

pad_mode is `'same'` :

$$
\begin{aligned}
D_{out} &= \left\lfloor \frac{D_{in}}{\text{stride}[0]} + 1 \right\rfloor \\
H_{out} &= \left\lfloor \frac{H_{in}}{\text{stride}[1]} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W_{in}}{\text{stride}[2]} + 1 \right\rfloor
\end{aligned}
$$

pad_mode is `'valid'` :

$$
\begin{aligned}
D_{out} &= \left\lfloor \frac{D_{in} - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1)}{\text{stride}[0]} + 1 \right\rfloor \\
H_{out} &= \left\lfloor \frac{H_{in} - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1)}{\text{stride}[1]} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W_{in} - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1)}{\text{stride}[2]} + 1 \right\rfloor
\end{aligned}
$$

pad_mode is `'pad'` :

$$
\begin{aligned}
D_{out} &= \left\lfloor \frac{D_{in} + padding[0] + padding[1] - (\text{dilation}[0] - 1) \times \text{kernel\_size}[0] - 1}{\text{stride}[0]} + 1 \right\rfloor \\
H_{out} &= \left\lfloor \frac{H_{in} + padding[2] + padding[3] - (\text{dilation}[1] - 1) \times \text{kernel\_size}[1] - 1}{\text{stride}[1]} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W_{in} + padding[4] + padding[5] - (\text{dilation}[2] - 1) \times \text{kernel\_size}[2] - 1}{\text{stride}[2]} + 1 \right\rfloor
\end{aligned}
$$

Raises

- **TypeError** – If *in_channels*, *out_channels* or *group* is not an int.
- **TypeError** – If *kernel_size*, *stride*, *padding* , *dilation* or *output_padding* is neither an int nor a tuple of three.
- **TypeError** – If input data type is not float16 or float32.
- **ValueError** – If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *padding* is less than 0.
- **ValueError** – If *pad_mode* is not one of `'same'`, `'valid'`, `'pad'`.
- **ValueError** – If *padding* is a tuple whose length is not equal to 6.
- **ValueError** – If *pad_mode* is not equal to `'pad'` and *padding* is not equal to (0, 0, 0, 0, 0, 0).
- **ValueError** – If *data_format* is not `'NCDHW'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.ones([32, 16, 10, 32, 32]), mindspore.float32)
>>> conv3d_transpose = nn.Conv3dTranspose(in_channels=16, out_channels=3, kernel_size=(4, 6,␣
↪2),
...                                       pad_mode='pad')
>>> output = conv3d_transpose(x)
>>> print(output.shape)
(32, 3, 13, 37, 33)
```

3.4.7 mindspore.nn.Unfold

class mindspore.nn.**Unfold**(*ksizes*, *strides*, *rates*, *padding='valid'*)

Extracts patches from images. The input tensor must be a 4-D tensor and the data format is NCHW.

Parameters

- **ksizes** (*Union[tuple[int], list[int]]*) –The size of sliding window, must be a tuple or a list of integers, and the format is [1, ksize_row, ksize_col, 1].
- **strides** (*Union[tuple[int], list[int]]*) –Distance between the centers of the two consecutive patches, must be a tuple or list of int, and the format is [1, stride_row, stride_col, 1].
- **rates** (*Union[tuple[int], list[int]]*) –In each extracted patch, the gap between the corresponding dimension pixel positions, must be a tuple or a list of integers, and the format is [1, rate_row, rate_col, 1].
- **padding** (*str*) –The type of padding algorithm, is a string whose value is `"same"` or `"valid"` , not case sensitive. Default: `"valid"` .
  - `"same"`: Means that the patch can take the part beyond the original image, and this part is filled with 0.
  - `"valid"`: Means that the taken patch area must be completely covered in the original image.

Inputs:

- **x** (Tensor) - A 4-D tensor whose shape is $[in\_batch, in\_depth, in\_row, in\_col]$ and data type is number.

Outputs:

Tensor, a 4-D tensor whose data type is same as *x*, and the shape is $(out\_batch, out\_depth, out\_row, out\_col)$ where *out_batch* is the same as the *in_batch*.

- $out\_depth = ksize\_row * ksize\_col * in\_depth$
- $out\_row = (in\_row - (ksize\_row + (ksize\_row - 1) * (rate\_row - 1)))//stride\_row + 1$
- $out\_col = (in\_col - (ksize\_col + (ksize\_col - 1) * (rate\_col - 1)))//stride\_col + 1$

Raises

- **TypeError** –If *ksizes*, *strides* or *rates* is neither a tuple nor list.
- **ValueError** –If shape of *ksizes*, *strides* or *rates* is not $(1, x\_row, x\_col, 1)$.
- **ValueError** –If the second and third element of *ksizes*, *strides* or *rates* is less than 1.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Unfold(ksizes=[1, 2, 2, 1], strides=[1, 2, 2, 1], rates=[1, 2, 2, 1])
>>> # As stated in the above code:
>>> # ksize_row = 2, ksize_col = 2, rate_row = 2, rate_col = 2, stride_row = 2, stride_col =␣
↪2.
>>> image = Tensor(np.ones([2, 3, 6, 6]), dtype=mindspore.float16)
>>> # in_batch = 2, in_depth = 3, in_row = 6, in_col = 6.
```

(continues on next page)

(continued from previous page)

```
>>> # Substituting the formula to get:
>>> # out_batch = in_batch = 2
>>> # out_depth = 2 * 2 * 3 = 12
>>> # out_row = (6 - (2 + (2 - 1) * (2 - 1))) // 2 + 1 = 2
>>> # out_col = (6 - (2 + (2 - 1) * (2 - 1))) // 2 + 1 = 2
>>> output = net(image)
>>> print(output.shape)
(2, 12, 2, 2)
```

3.5 Recurrent Layer

API Name	Description	Supported Platforms
mindspore.nn.RNN	Stacked Elman RNN layers, applying RNN layer with tanh or ReLU non-linearity to the input.	Ascend GPU CPU
mindspore.nn.RNNCell	An Elman RNN cell with tanh or ReLU non-linearity.	Ascend GPU CPU
mindspore.nn.GRU	Stacked GRU (Gated Recurrent Unit) layers.	Ascend GPU CPU
mindspore.nn.GRUCell	A GRU(Gated Recurrent Unit) cell.	Ascend GPU CPU
mindspore.nn.LSTM	Stacked LSTM (Long Short-Term Memory) layers.	Ascend GPU CPU
mindspore.nn.LSTMCell	A LSTM (Long Short-Term Memory) cell.	Ascend GPU CPU

3.5.1 mindspore.nn.RNN

class mindspore.nn.**RNN**(**args*, ***kwargs*)

Stacked Elman RNN layers, applying RNN layer with tanh or ReLU non-linearity to the input.

For each element in the input sequence, each layer computes the following function:

$$h_t = activation(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

Here h_t is the hidden state at time t, x_t is the input at time t, and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time *0*. W_{ih} is the learnable input-hidden weights, and b_{ih} is the learnable input-hidden bias. W_{hh} is the learnable hidden-hidden weights, and b_{hh} is the learnable hidden-hidden bias.

Parameters

- **input_size** (*int*) –Number of features of input.
- **hidden_size** (*int*) –Number of features of hidden layer.
- **num_layers** (*int*) –Number of layers of stacked RNN. Default: `1` .
- **nonlinearity** (*str*) –The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`.
- **has_bias** (*bool*) –Whether the cell has bias b_{ih} and b_{hh}. Default: `True` .
- **batch_first** (*bool*) –Specifies whether the first dimension of input *x* is batch_size. Default: `False` .
- **dropout** (*float*) –If not 0.0, append *Dropout* layer on the outputs of each RNN layer except the last layer. Default `0.0` . The range of dropout is [0.0, 1.0).
- **bidirectional** (*bool*) –Specifies whether it is a bidirectional RNN, num_directions=2 if bidirectional=True otherwise 1. Default: `False` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of data type mindspore.float32 or mindspore.float16 and shape $(seq\_len, batch\_size, input\_size)$ or $(batch\_size, seq\_len, input\_size)$.
- **hx** (Tensor) - Tensor of data type mindspore.float32 or mindspore.float16 and shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.
- **seq_length** (Tensor) - The length of each sequence in an input batch. Tensor of shape $(batch\_size)$. Default: `None` . This input indicates the real sequence length before padding to avoid padded elements have been used to compute hidden state and affect the final output. It is recommended to use this input when *x* has padding elements.

Outputs:

Tuple, a tuple contains (*output*, *hx_n*).

- **output** (Tensor) - Tensor of shape $(seq\_len, batch\_size, num\_directions * hidden\_size)$ or $(batch\_size, seq\_len, num\_directions * hidden\_size)$.
- **hx_n** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.

Raises

- **TypeError** – If *input_size*, *hidden_size* or *num_layers* is not an int.
- **TypeError** – If *has_bias*, *batch_first* or *bidirectional* is not a bool.
- **TypeError** – If *dropout* is not a float.
- **ValueError** – If *dropout* is not in range [0.0, 1.0).
- **ValueError** – If *nonlinearity* is not in ['tanh', 'relu'].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.RNN(10, 16, 2, has_bias=True, batch_first=True, bidirectional=False)
>>> x = ms.Tensor(np.ones([3, 5, 10]).astype(np.float32))
>>> h0 = ms.Tensor(np.ones([1 * 2, 3, 16]).astype(np.float32))
>>> output, hn = net(x, h0)
>>> print(output.shape)
(3, 5, 16)
```

3.5.2 mindspore.nn.RNNCell

class mindspore.nn.**RNNCell**(*input_size: int*, *hidden_size: int*, *has_bias: bool = True*, *nonlinearity: str = 'tanh'*, *dtype=mstype.float32*)

An Elman RNN cell with tanh or ReLU non-linearity.

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

Here h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time *0*. If *nonlinearity* is *relu*, then *relu* is used instead of *tanh*.

Parameters

- **input_size** (*int*) –Number of features of input.
- **hidden_size** (*int*) –Number of features of hidden layer.
- **has_bias** (*bool*) –Whether the cell has bias b_{ih} and b_{hh}. Default: `True` .
- **nonlinearity** (*str*) –The non-linearity to use. Can be either `"tanh"` or `"relu"` . Default: `"tanh"` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(batch\_size, input\_size)$.
- **hx** (Tensor) - Tensor of data type mindspore.float32 and shape $(batch\_size, hidden\_size)$.

Outputs:

- **hx'** (Tensor) - Tensor of shape $(batch\_size, hidden\_size)$.

Raises

- **TypeError** –If *input_size* or *hidden_size* is not an int or not greater than 0.
- **TypeError** –If *has_bias* is not a bool.
- **ValueError** –If *nonlinearity* is not in ['tanh', 'relu'].

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.RNNCell(10, 16)
>>> x = ms.Tensor(np.ones([5, 3, 10]).astype(np.float32))
>>> hx = ms.Tensor(np.ones([3, 16]).astype(np.float32))
>>> output = []
>>> for i in range(5):
...     hx = net(x[i], hx)
...     output.append(hx)
>>> print(output[0].shape)
(3, 16)
```

3.5.3 mindspore.nn.GRU

class mindspore.nn.**GRU**(**args*, ***kwargs*)

Stacked GRU (Gated Recurrent Unit) layers.

Apply GRU layer to the input.

There are two gates in a GRU model. One is update gate and the other is reset gate. Denote two consecutive time nodes as $t-1$ and t. Given an input x_t at time t, a hidden state h_{t-1}, the update and reset gate at time t is computed using a gating mechanism. Update gate z_t is designed to protect the cell from perturbation by irrelevant inputs and past hidden state. Reset gate r_t determines how much information should be reset from old hidden state. New memory state n_t is calculated with the current input, on which

the reset gate will be applied. Finally, current hidden state h_t is computed with the calculated update grate and new memory state. The complete formulation is as follows:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

Here σ is the sigmoid function, and $*$ is the Hadamard product. W, b are learnable weights between the output and the input in the formula. For instance, W_{ir}, b_{ir} are the weight and bias used to transform from input x to r. Details can be found in paper Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.

Note: When using GRU on Ascend, the hidden size only supports multiples of 16.

Parameters

- **input_size** (*int*) −Number of features of input.
- **hidden_size** (*int*) −Number of features of hidden layer.
- **num_layers** (*int*) −Number of layers of stacked GRU. Default: `1` .
- **has_bias** (*bool*) −Whether the cell has bias b_{in} and b_{hn}. Default: `True` .
- **batch_first** (*bool*) −Specifies whether the first dimension of input *x* is batch_size. Default: `False` .
- **dropout** (*float*) −If not 0.0, append *Dropout* layer on the outputs of each GRU layer except the last layer. Default `0.0` . The range of dropout is [0.0, 1.0).
- **bidirectional** (*bool*) −Specifies whether it is a bidirectional GRU, num_directions=2 if bidirectional=True otherwise 1. Default: `False` .
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of data type mindspore.float32 or mindspore.float16 and shape $(seq\_len, batch\_size, input\_size)$ or $(batch\_size, seq\_len, input\_size)$.
- **hx** (Tensor) - Tensor of data type mindspore.float32 or mindspore.float16 and shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.
- **seq_length** (Tensor) - The length of each sequence in an input batch. Tensor of shape (batch_size). Default: `None` . This input indicates the real sequence length before padding to avoid padded elements have been used to compute hidden state and affect the final output. It is recommended to use this input when **x** has padding elements.

Outputs:

Tuple, a tuple contains (*output*, *h_n*).

- **output** (Tensor) - Tensor of shape $(seq\_len, batch\_size, num\_directions * hidden\_size)$ or $(batch\_size, seq\_len, num\_directions * hidden\_size)$.
- **hx_n** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.

Raises

- **TypeError** −If *input_size*, *hidden_size* or *num_layers* is not an int.
- **TypeError** −If *has_bias*, *batch_first* or *bidirectional* is not a bool.
- **TypeError** −If *dropout* is not a float.

- **ValueError** –If *dropout* is not in range [0.0, 1.0).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.GRU(10, 16, 2, has_bias=True, batch_first=True, bidirectional=False)
>>> x = ms.Tensor(np.ones([3, 5, 10]).astype(np.float32))
>>> h0 = ms.Tensor(np.ones([1 * 2, 3, 16]).astype(np.float32))
>>> output, hn = net(x, h0)
>>> print(output.shape)
(3, 5, 16)
```

3.5.4 mindspore.nn.GRUCell

class mindspore.nn.**GRUCell**(*input_size: int*, *hidden_size: int*, *has_bias: bool = True*, *dtype=mstype.float32*)

A GRU(Gated Recurrent Unit) cell.

$$
\begin{aligned}
r &= \sigma(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \\
z &= \sigma(W_{iz}x + b_{iz} + W_{hz}h + b_{hz}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - z) * n + z * h
\end{aligned}
$$

Here σ is the sigmoid function, and $*$ is the Hadamard product. W, b are learnable weights between the output and the input in the formula. h is hidden state. r is reset gate. z is update gate. n is n-th layer. For instance, W_{ir}, b_{ir} are the weight and bias used to transform from input x to r. Details can be found in paper Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.

Parameters

- **input_size** (*int*) –Number of features of input.
- **hidden_size** (*int*) –Number of features of hidden layer.
- **has_bias** (*bool*) –Whether the cell has bias b_{in} and b_{hn}. Default: `True` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(batch\_size, input\_size)$.
- **hx** (Tensor) - Tensor of data type mindspore.float32 and shape $(batch\_size, hidden\_size)$.

Outputs:

- **hx'** (Tensor) - Tensor of shape $(batch\_size, hidden\_size)$.

Raises

- **TypeError** –If *input_size*, *hidden_size* is not an int.
- **TypeError** –If *has_bias* is not a bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.GRUCell(10, 16)
>>> x = ms.Tensor(np.ones([5, 3, 10]).astype(np.float32))
>>> hx = ms.Tensor(np.ones([3, 16]).astype(np.float32))
>>> output = []
>>> for i in range(5):
...     hx = net(x[i], hx)
...     output.append(hx)
>>> print(output[0].shape)
(3, 16)
```

3.5.5 mindspore.nn.LSTM

class mindspore.nn.**LSTM**(**args*, ***kwargs*)

Stacked LSTM (Long Short-Term Memory) layers.

Apply LSTM layer to the input.

There are two pipelines connecting two consecutive cells in a LSTM model; one is cell state pipeline and the other is hidden state pipeline. Denote two consecutive time nodes as $t-1$ and t. Given an input x_t at time t, an hidden state h_{t-1} and an cell state c_{t-1} of the layer at time $t-1$, the cell state and hidden state at time t is computed using an gating mechanism. Input gate i_t is designed to protect the cell from perturbation by irrelevant inputs. Forget gate f_t affords protection of the cell by forgetting some information in the past, which is stored in h_{t-1}. Output gate o_t protects other units from perturbation by currently irrelevant memory contents. Candidate cell state $\tilde{c}_t$ is calculated with the current input, on which the input gate will be applied. Finally, current cell state c_t and hidden state h_t are computed with the calculated gates and cell states. The complete formulation is as follows.

$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + b_{ix} + W_{ih}h_{(t-1)} + b_{ih}) \\
f_t &= \sigma(W_{fx}x_t + b_{fx} + W_{fh}h_{(t-1)} + b_{fh}) \\
\tilde{c}_t &= \tanh(W_{cx}x_t + b_{cx} + W_{ch}h_{(t-1)} + b_{ch}) \\
o_t &= \sigma(W_{ox}x_t + b_{ox} + W_{oh}h_{(t-1)} + b_{oh}) \\
c_t &= f_t * c_{(t-1)} + i_t * \tilde{c}_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

Here σ is the sigmoid function, and $*$ is the Hadamard product. W, b are learnable weights between the output and the input in the formula. For instance, W_{ix}, b_{ix} are the weight and bias used to transform from input x to i. Details can be found in paper LONG SHORT-TERM MEMORY and Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling.

LSTM hides the cycle of the whole cyclic neural network on the time step of the sequence, and input the sequence and initial state to obtain the matrix spliced by the hidden state of each time step and the hidden state of the last time step. We use the hidden state of the last time step as the coding feature of the input sentence and output it to the next layer.

$$h_{0:n}, (h_n, c_n) = LSTM(x_{0:n}, (h_0, c_0))$$

Parameters

- **input_size** (*int*) –Number of features of input.
- **hidden_size** (*int*) –Number of features of hidden layer.

- **num_layers** (*int*) −Number of layers of stacked LSTM . Default: `1` .
- **has_bias** (*bool*) −Whether the cell has bias b_{ih} and b_{fh}. Default: `True` .
- **batch_first** (*bool*) −Specifies whether the first dimension of input *x* is batch_size. Default: `False` .
- **dropout** (*float, int*) −If not 0, append *Dropout* layer on the outputs of each LSTM layer except the last layer. Default `0` . The range of dropout is [0.0, 1.0).
- **bidirectional** (*bool*) −Specifies whether it is a bidirectional LSTM, num_directions=2 if bidirectional=True otherwise 1. Default: `False` .
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of data type mindspore.float32 or mindspore.float16 and shape ($seq\_len, batch\_size, input\_size$) or ($batch\_size, seq\_len, input\_size$) .
- **hx** (tuple) - A tuple of two Tensors (h_0, c_0) both of data type mindspore.float32 or mindspore.float16 and shape ($num\_directions * num\_layers, batch\_size, hidden\_size$) .
- **seq_length** (Tensor) - The length of each sequence in an input batch. Tensor of shape ($batch\_size$). Default: `None` . This input indicates the real sequence length before padding to avoid padded elements have been used to compute hidden state and affect the final output. It is recommended to use this input when **x** has padding elements.

Outputs:

Tuple, a tuple contains (*output*, (*h_n*, *c_n*)).

- **output** (Tensor) - Tensor of shape ($seq\_len, batch\_size, num\_directions * hidden\_size$) .
- **hx_n** (tuple) - A tuple of two Tensor (h_n, c_n) both of shape ($num\_directions * num\_layers, batch\_size, hidden\_size$) .

Raises

- **TypeError** −If *input_size*, *hidden_size* or *num_layers* is not an int.
- **TypeError** −If *has_bias*, *batch_first* or *bidirectional* is not a bool.
- **TypeError** −If *dropout* is not a float.
- **ValueError** −If *dropout* is not in range [0.0, 1.0).

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.LSTM(10, 16, 2, has_bias=True, batch_first=True, bidirectional=False)
>>> x = ms.Tensor(np.ones([3, 5, 10]).astype(np.float32))
>>> h0 = ms.Tensor(np.ones([1 * 2, 3, 16]).astype(np.float32))
>>> c0 = ms.Tensor(np.ones([1 * 2, 3, 16]).astype(np.float32))
>>> output, (hn, cn) = net(x, (h0, c0))
>>> print(output.shape)
(3, 5, 16)
```

3.5.6 mindspore.nn.LSTMCell

class mindspore.nn.**LSTMCell**(*input_size: int*, *hidden_size: int*, *has_bias: bool = True*, *dtype=mstype.float32*)

A LSTM (Long Short-Term Memory) cell.

$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + b_{ix} + W_{ih}h_{(t-1)} + b_{ih}) \\
f_t &= \sigma(W_{fx}x_t + b_{fx} + W_{fh}h_{(t-1)} + b_{fh}) \\
\tilde{c}_t &= \tanh(W_{cx}x_t + b_{cx} + W_{ch}h_{(t-1)} + b_{ch}) \\
o_t &= \sigma(W_{ox}x_t + b_{ox} + W_{oh}h_{(t-1)} + b_{oh}) \\
c_t &= f_t * c_{(t-1)} + i_t * \tilde{c}_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

Here σ is the sigmoid function, and $*$ is the Hadamard product. W, b are learnable weights between the output and the input in the formula. For instance, W_{ix}, b_{ix} are the weight and bias used to transform from input x to i. Details can be found in paper LONG SHORT-TERM MEMORY and Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling.

The encapsulated LSTMCell can be simplified to the following formula:

$$h^{'}, c^{'} = LSTMCell(x, (h_0, c_0))$$

Parameters

- **input_size** (*int*) −Number of features of input.
- **hidden_size** (*int*) −Number of features of hidden layer.
- **has_bias** (*bool*) −Whether the cell has bias *b_{ih}* and *b_{hh}*. Default: `True` .
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(batch\_size, input\_size)$.
- **hx** (tuple) - A tuple of two Tensors (h_0, c_0) both of data type mindspore.float32 and shape $(batch\_size, hidden\_size)$.

Outputs:

- **hx'** (Tensor) - A tuple of two Tensors (h', c') both of data shape $(batch\_size, hidden\_size)$.

Raises

- **TypeError** −If *input_size*, *hidden_size* is not an int.
- **TypeError** −If *has_bias* is not a bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.LSTMCell(10, 16)
>>> x = ms.Tensor(np.ones([5, 3, 10]).astype(np.float32))
>>> h = ms.Tensor(np.ones([3, 16]).astype(np.float32))
>>> c = ms.Tensor(np.ones([3, 16]).astype(np.float32))
>>> output = []
>>> for i in range(5):
...     hx = net(x[i], (h, c))
...     output.append(hx)
>>> print(output[0][0].shape)
(3, 16)
```

3.6 Transformer Layer

API Name	Description	Supported Platforms
mindspore.nn.MultiheadAttention	This is an implementation of multihead attention in the paper Attention is all you need.	Ascend GPU CPU
mindspore.nn.TransformerEncoderLayer	Transformer Encoder Layer.	Ascend GPU CPU
mindspore.nn.TransformerDecoderLayer	Transformer Decoder Layer.	Ascend GPU CPU
mindspore.nn.TransformerEncoder	Transformer Encoder module with multi-layer stacked of *mindspore.nn.TransformerEncoderLayer*, including multihead attention and feedforward layer.	Ascend GPU CPU
mindspore.nn.TransformerDecoder	Transformer Decoder module with multi-layer stacked of *mindspore.nn.TransformerDecoderLayer*, including multihead self attention, cross attention and feedforward layer.	Ascend GPU CPU
mindspore.nn.Transformer	Transformer module including encoder and decoder.	Ascend GPU CPU

3.6.1 mindspore.nn.MultiheadAttention

class mindspore.nn.**MultiheadAttention**(*embed_dim*, *num_heads*, *dropout=0.0*, *has_bias=True*, *add_bias_kv=False*, *add_zero_attn=False*, *kdim=None*, *vdim=None*, *batch_first=False*, *dtype=mstype.float32*)

This is an implementation of multihead attention in the paper Attention is all you need. Given the query vector, the key vector and value vector, the attention will be performed as the following:

$$MultiHeadAttention(query, key, value) = Concat(head_1, \ldots, head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$, and W^O , W_i^Q , W_i^K , W_i^V are weight matrices. The default input / output projection layers is with a bias.

if query, key and value tensor is same, then it will be self attention.

Parameters

- **embed_dim** (*int*) −Total dimension of MultiheadAttention.
- **num_heads** (*int*) −Number of attention heads. Note that *embed_dim* will be split across *num_heads* (i.e. each head will have dimension *embed_dim // num_heads*).
- **dropout** (*float, optional*) −Dropout probability of *attn_output_weights*. Default: `0.0`.
- **has_bias** (*bool, optional*) −Whether adds bias to input / output projection layers. Default: `True`.
- **add_bias_kv** (*bool, optional*) −Whether adds bias to the key and value sequences at axis=0. Default: `False`.
- **add_zero_attn** (*bool, optional*) −Whether adds a new batch of zeros to the key and value sequences at axis=1. Default: `False`.
- **kdim** (*int, optional*) −Total number of features for keys. Default: `None` (*kdim=embed_dim*).
- **vdim** (*int, optional*) −Total number of features for values. Default: `None` (*vdim=embed_dim*).
- **batch_first** (*bool, optional*) −If `True`, then the input and output shape are $(batch, seq, feature)$, else $(seq, batch, feature)$. Default: `False`.
- **dtype** (*mindspore.dtype*, optional) −Data type of Parameter. Default: `mstype.float32` .

Inputs:

- **query** (Tensor) - The query embeddings. If *query* is unbatched, the shape is (L, E_q), otherwise the shape is (L, N, E_q) when *batch_first=False* or (N, L, E_q) when *batch_first=True* , where L is the batch size, and E_q is the query embedding dimension *embed_dim*. Supported types: float16, float32, float64. Queries are compared against key-value pairs to produce the output.
- **key** (Tensor) - The key embeddings. If *key* is unbatched, the shape is (S, E_k), otherwise the shape is (S, N, E_k) when *batch_first=False* or (N, S, E_k) when *batch_first=True* , where S is the source sequence length, N is the batch size, and E_k is the key embedding dimension *kdim*. Supported types: float16, float32, float64.
- **value** (Tensor) - The value embeddings. If *value* is unbatched, the shape is (S, E_v), otherwise the shape is (S, N, E_v) when *batch_first=False* or (N, S, E_v) when *batch_first=True* , where S is the source sequence length, N is the batch size, and E_v is the value embedding dimension *vdim*. Supported types: float16, float32, float64.
- **key_padding_mask** (Tensor, optional) - If specified, a mask of shape (N, S) indicating which elements within *key* to ignore for the purpose of attention (i.e. treat as "padding"). For unbatched *query*, shape should be (S). Binary and float masks are supported. For a binary mask, a `True` value indicates that the corresponding *key* value will be ignored for the purpose of attention. For a float mask, it will be directly added to the corresponding *key* value. Supported float types: float16, float32, float64. Default: `None`.
- **need_weights** (bool, optional) - Whether returns *attn_output_weights* in addition to *attn_outputs*. Default: `True`.
- **attn_mask** (Tensor, optional) - If specified, a 2D or 3D mask preventing attention to certain positions. Must be of shape (L, S) or $(N \cdot \text{num\_heads}, L, S)$, where N is the batch size, L is the target sequence length, and S is the source sequence length. A 2D mask will be broadcasted across the batch while a 3D mask allows for a different mask for each entry in the batch. For a binary mask, a `True` value indicates that the corresponding position is not allowed to attend. For a float mask, the mask values will be added to the attention weight. Supported float types: float16, float32, float64. Default: `None`.
- **average_attn_weights** (bool, optional) - If true, indicates that the returned *attn_weights* should be averaged across heads. Otherwise, *attn_weights* are provided separately per head. Note that this flag only has an effect when *need_weights=True*. Default: `True` (i.e. average weights across heads)

Outputs:

Tuple, a tuple contains(*attn_output*, *attn_output_weights*)

- **attn_output** - Attention outputs. If input is unbatched, the output shape is (L, E), otherwise the output shape is (L, N, E) when *batch_first=False* or (N, L, E) when *batch_first=True* , where L is the target sequence length, N is the batch size, and E is the embedding dimension *embed_dim*.
- **attn_output_weights** - Only returned when *need_weights=True*. If *average_attn_weights=True*, returns attention weights averaged across heads with shape (L, S) when input is unbatched or (N, L, S) when input is batched, where N is the batch size, L is the target sequence length, and S is the source sequence length. If *average_attn_weights=False*, returns attention weights per head of shape (num_heads, L, S) when input is unbatched or (N, num_heads, L, S) when input is batched.

Raises

- **ValueError** –If the init argument *embed_dim* is not divisible by *num_heads*.
- **TypeError** –If the input argument *key_padding_mask* is not bool or floating types.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> embed_dim, num_heads = 128, 8
>>> seq_length, batch_size = 10, 8
>>> query = ms.Tensor(np.random.randn(seq_length, batch_size, embed_dim), ms.float32)
>>> key = ms.Tensor(np.random.randn(seq_length, batch_size, embed_dim), ms.float32)
>>> value = ms.Tensor(np.random.randn(seq_length, batch_size, embed_dim), ms.float32)
>>> multihead_attn = ms.nn.MultiheadAttention(embed_dim, num_heads)
>>> attn_output, attn_output_weights = multihead_attn(query, key, value)
>>> print(attn_output.shape)
(10, 8, 128)
```

3.6.2 mindspore.nn.TransformerEncoderLayer

class mindspore.nn.**TransformerEncoderLayer**(*d_model: int*, *nhead: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*, *activation: Union[str, Cell, callable] = 'relu'*, *layer_norm_eps: float = 1e-5*, *batch_first: bool = False*, *norm_first: bool = False*, *dtype=mstype.float32*)

Transformer Encoder Layer. This is an implementation of the single layer of the transformer encoder layer, including multihead attention and feedward layer.

Parameters

- **d_model** (*int*) –The number of features in the input tensor.
- **nhead** (*int*) –The number of heads in the MultiheadAttention modules.
- **dim_feedforward** (*int*) –The dimension of the feedforward layer. Default: `2048`.
- **dropout** (*float*) –The dropout value. Default: `0.1`.
- **activation** (*Union[str, callable, Cell]*) –The activation function of the intermediate layer, can be a string (`"relu"` or `"gelu"`), Cell instance (*mindspore.nn.ReLU* or *mindspore.nn.GELU*) or a callable (*mindspore.ops.relu()* or *mindspore.ops.gelu()*). Default: `"relu"`.
- **layer_norm_eps** (*float*) –The epsilon value in LayerNorm modules. Default: `1e-5`.

- **batch_first** (*bool*) –If *batch_first=True* , then the shape of input and output tensors is $(batch, seq, feature)$, otherwise the shape is $(seq, batch, feature)$. Default: `False`.
- **norm_first** (*bool*) –If *norm_first = True*, layer norm is located prior to attention and feedforward operations; if *norm_first = False*, layer norm is located after the attention and feedforward operations. Default: `False`.
- **dtype** (*mindspore.dtype*) –Data type of Parameter. Default: `mstype.float32` .

Inputs:

- **src** (Tensor) - the sequence to the encoder layer. For unbatched input, the shape is (S, E) ; otherwise if *batch_first=False* , the shape is (S, N, E) and if *batch_first=True* , the shape is (N, S, E), where (S) is the source sequence length, (N) is the batch number and (E) is the feature number. Supported types: float16, float32, float64.
- **src_mask** (Tensor, optional) - the mask for the src sequence. The shape is (S, S) or $(N * nhead, S, S)$. Supported types: float16, float32, float64, bool. Default: `None`.
- **src_key_padding_mask** (Tensor, optional) - the mask for the src keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.

Outputs:

Tensor. The shape and dtype of Tensor is the same with *src* .

Raises

- **ValueError** –If the init argument *activation* is not str, callable or Cell instance.
- **ValueError** –If the init argument *activation* is not *mindspore.nn.ReLU*, *mindspore.nn.GELU* instance, *mindspore.ops.relu()*, *mindspore.ops.gelu()*, "relu" or "gelu" .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> encoder_layer = ms.nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> src = ms.Tensor(np.random.rand(10, 32, 512), ms.float32)
>>> out = encoder_layer(src)
>>> print(out.shape)
(10, 32, 512)
>>> # Alternatively, when batch_first=True:
>>> encoder_layer = ms.nn.TransformerEncoderLayer(d_model=512, nhead=8, batch_first=True)
>>> src = ms.Tensor(np.random.rand(32, 10, 512), ms.float32)
>>> out = encoder_layer(src)
>>> print(out.shape)
(32, 10, 512)
```

3.6.3 mindspore.nn.TransformerDecoderLayer

class mindspore.nn.**TransformerDecoderLayer**(*d_model: int*, *nhead: int*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*, *activation: Union[str, Cell, callable] = 'relu'*, *layer_norm_eps: float = 1e-5*, *batch_first: bool = False*, *norm_first: bool = False*, *dtype=mstype.float32*)

Transformer Decoder Layer. This is an implementation of the single layer of the transformer decoder layer, including self-attention, cross attention and feedward layer.

Parameters

- **d_model** (*int*) –The number of expected features in the input tensor.
- **nhead** (*int*) –The number of heads in the MultiheadAttention modules.
- **dim_feedforward** (*int*) –The dimension of the feedforward layer. Default: `2048`.
- **dropout** (*float*) –The dropout value. Default: `0.1`.
- **activation** (*Union[str, callable, Cell]*) –The activation function of the intermediate layer, can be a string (`"relu"` or `"gelu"`), Cell instance (*mindspore.nn.ReLU* or *mindspore.nn.GELU*) or a callable (*mindspore.ops.relu()* or *mindspore.ops.gelu()*). Default: `"relu"`.
- **layer_norm_eps** (*float*) –The epsilon value in LayerNorm modules. Default: `1e-5`.
- **batch_first** (*bool*) –If *batch_first=True* , then the shape of input and output tensors is $(batch, seq, feature)$, otherwise the shape is $(seq, batch, feature)$. Default: `False`.
- **norm_first** (*bool*) –If *norm_first = True*, layer norm is located prior to attention and feedforward operations; if *norm_first = False*, layer norm is located after the attention and feedforward operations. Default: `False`.
- **dtype** (*mindspore.dtype*) –Data type of Parameter. Default: `mstype.float32` .

Inputs:

- **tgt** (Tensor) - The sequence to the decoder layer. For unbatched input, the shape is (T, E) ; otherwise if *batch_first=False* , the shape is (T, N, E) and if *batch_first=True* , the shape is (N, T, E), where (T) is the target sequence length. Supported types: float16, float32, float64.
- **memory** (Tensor) - The sequence from the last layer of the encoder. Supported types: float16, float32, float64.
- **tgt_mask** (Tensor, optional) - The mask of the tgt sequence. The shape is (T, T) or $(N * nhead, T, T)$. Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_mask** (Tensor, optional) - The mask of the memory sequence. The shape is (T, S) . Supported types: float16, float32, float64, bool. Default: `None`.
- **tgt_key_padding_mask** (Tensor, optional): The mask of the tgt keys per batch. The shape is (T) for unbatched input, otherwise (N, T) . Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_key_padding_mask** (Tensor, optional) - The mask of the memory keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.

Outputs:

Tensor. The shape and dtype of Tensor is the same with *tgt* .

Raises

- **ValueError** –If the init argument *activation* is not str, callable or Cell instance.
- **ValueError** –If the init argument *activation* is not *mindspore.nn.ReLU*, *mindspore.nn.GELU* instance, *mindspore.ops.relu()*, *mindspore.ops.gelu()* , "relu" or "gelu" .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> decoder_layer = ms.nn.TransformerDecoderLayer(d_model=512, nhead=8)
>>> memory = ms.Tensor(np.random.rand(10, 32, 512), ms.float32)
>>> tgt = ms.Tensor(np.random.rand(20, 32, 512), ms.float32)
>>> out = decoder_layer(tgt, memory)
>>> print(out.shape)
(20, 32, 512)
>>> # Alternatively, when `batch_first` is ``True``:
>>> decoder_layer = ms.nn.TransformerDecoderLayer(d_model=512, nhead=8, batch_first=True)
>>> memory = ms.Tensor(np.random.rand(32, 10, 512), ms.float32)
>>> tgt = ms.Tensor(np.random.rand(32, 20, 512), ms.float32)
>>> out = decoder_layer(tgt, memory)
>>> print(out.shape)
(32, 20, 512)
```

3.6.4 mindspore.nn.TransformerEncoder

class mindspore.nn.**TransformerEncoder**(*encoder_layer*, *num_layers*, *norm=None*)

Transformer Encoder module with multi-layer stacked of *mindspore.nn.TransformerEncoderLayer*, including multihead attention and feedforward layer. Users can build the BERT(https://arxiv.org/abs/1810.04805) model with corresponding parameters.

Parameters

- **encoder_layer** (Cell) –An instance of the *mindspore.nn.TransformerEncoderLayer* class.
- **num_layers** (*int*) –The number of encoder-layers in the encoder.
- **norm** (Cell, *optional*) –The layer normalization module. Default: None.

Inputs:

- **src** (Tensor) - The sequence to the encoder. For unbatched input, the shape is (S, E) ; otherwise if *batch_first=False* in *mindspore.nn.TransformerEncoderLayer*, the shape is (S, N, E) and if *batch_first=True* , the shape is (N, S, E), where (S) is the source sequence length, (N) is the batch number and (E) is the feature number. Supported types: float16, float32, float64.
- **src_mask** (Tensor, optional) - The mask of the src sequence. The shape is (S, S) or $(N * nhead, S, S)$, where *nhead* is the arguent in *mindspore.nn.TransformerEncoderLayer*. Supported types: float16, float32, float64, bool. Default: None.
- **src_key_padding_mask** (Tensor, optional) - the mask of the src keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: None.

Outputs:

Tensor. The shape and dtype of Tensor is the same with *src* .

Raises

AssertionError –If the input argument *src_key_padding_mask* is not bool or floating types.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> encoder_layer = ms.nn.TransformerEncoderLayer(d_model=512, nhead=8)
>>> transformer_encoder = ms.nn.TransformerEncoder(encoder_layer, num_layers=6)
>>> src = ms.Tensor(np.random.rand(10, 32, 512), ms.float32)
>>> out = transformer_encoder(src)
>>> print(out.shape)
(10, 32, 512)
```

3.6.5 mindspore.nn.TransformerDecoder

class mindspore.nn.**TransformerDecoder**(*decoder_layer*, *num_layers*, *norm=None*)

Transformer Decoder module with multi-layer stacked of *mindspore.nn.TransformerDecoderLayer*, including multihead self attention, cross attention and feedforward layer.

Parameters

- **decoder_layer** (`Cell`) – An instance of the *mindspore.nn.TransformerDecoderLayer* class.
- **num_layers** (*int*) – The number of decoder-layers in the decoder.
- **norm** (`Cell, optional`) – The layer normalization module. Default: `None`.

Inputs:

- **tgt** (Tensor) - The sequence to the decoder. For unbatched input, the shape is (T, E) ; otherwise if *batch_first=False* in *mindspore.nn.TransformerDecoderLayer*, the shape is (T, N, E) and if *batch_first=True* , the shape is (N, T, E), where (T) is the target sequence length, (N) is the number of batches, and (E) is the number of features. Supported types: float16, float32, float64.
- **memory** (Tensor) - The sequence from the last layer of the encoder. Supported types: float16, float32, float64.
- **tgt_mask** (Tensor, optional) - the mask of the tgt sequence. The shape is (T, T) or $(N * nhead, T, T)$, where *nhead* is the arguent in *mindspore.nn.TransformerDecoderLayer*. Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_mask** (Tensor, optional) - the mask of the memory sequence. The shape is (T, S) . Supported types: float16, float32, float64, bool. Default: `None`.
- **tgt_key_padding_mask** (Tensor, optional) - the mask of the tgt keys per batch. Shape is (T). Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_key_padding_mask** (Tensor, optional) - the mask of the memory keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.

Outputs:

Tensor. The shape and dtype of Tensor is the same with *tgt* .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> decoder_layer = ms.nn.TransformerDecoderLayer(d_model=512, nhead=8)
>>> transformer_decoder = ms.nn.TransformerDecoder(decoder_layer, num_layers=6)
>>> memory = ms.Tensor(np.random.rand(10, 32, 512), ms.float32)
>>> tgt = ms.Tensor(np.random.rand(20, 32, 512), ms.float32)
>>> out = transformer_decoder(tgt, memory)
>>> print(out.shape)
(20, 32, 512)
```

3.6.6 mindspore.nn.Transformer

class mindspore.nn.**Transformer**(*d_model: int = 512*, *nhead: int = 8*, *num_encoder_layers: int = 6*, *num_decoder_layers: int = 6*, *dim_feedforward: int = 2048*, *dropout: float = 0.1*, *activation: Union[str, Cell, callable] = 'relu'*, *custom_encoder: Optional[Cell] = None*, *custom_decoder: Optional[Cell] = None*, *layer_norm_eps: float = 1e-5*, *batch_first: bool = False*, *norm_first: bool = False*, *dtype=mstype.float32*)

Transformer module including encoder and decoder. The difference with the original implements is the module use the residual addition before the layer normalization. And the default hidden activation is *gelu*. The details can be found in Attention is all you need.

Parameters

- **d_model** (*int*) –The number of expected features in the inputs tensor for Encoder and Decoder. Default: `512`.
- **nhead** (*int*) –The number of heads in the MultiheadAttention modules. Default: `8`.
- **num_encoder_layers** (*int*) –The number of encoder-layers in the encoder. Default: `6`.
- **num_decoder_layers** (*int*) –The number of decoder-layers in the decoder. Default: `6`.
- **dim_feedforward** (*int*) –The dimension of the feedforward layer. Default: `2048`.
- **dropout** (*float*) –The dropout value. Default: `0.1`.
- **activation** (*Union[str, callable, Cell]*) –The activation function of the intermediate layer, can be a string (`"relu"` or `"gelu"`), Cell instance (*mindspore.nn.ReLU* or *mindspore.nn.GELU*) or a callable (*mindspore.ops.relu()* or *mindspore.ops.gelu()*). Default: `"relu"`.
- **custom_encoder** (*Cell*) –Custom encoder. Default: `None`.
- **custom_decoder** (*Cell*) –Custom decoder. Default: `None`.
- **layer_norm_eps** (*float*) –the epsilion value in layer normalization module. Default: `1e-5`.
- **batch_first** (*bool*) –If *batch_first=True*, then the shape of input and output tensors is $(batch, seq, feature)$, otherwise the shape is $(seq, batch, feature)$. Default: `False`.
- **norm_first** (*bool*) –If *norm_first = True*, layer norm is located prior to attention and feedforward operations; if *norm_first = False*, layer norm is located after the attention and feedforward operations. Default: `False`.
- **dtype** (*mindspore.dtype*) –Data type of Parameter. Default: `mstype.float32` .

Inputs:

- **src** (Tensor) - The source sequence to the encoder. For unbatched input, the shape is (S, E) ; otherwise if *batch_first=False* , the shape is (S, N, E) and if *batch_first=True* , the shape is (N, S, E), where (S) is the source sequence length, (N) is the batch number and (E) is the feature number. Supported types: float16, float32, float64.

- **tgt** (Tensor) - The target sequence to the decoder. For unbatched input, the shape is (T, E) ; otherwise if *batch_first=False* , the shape is (T, N, E) and if *batch_first=True* , the shape is (N, T, E), where (T) is the target sequence length. Supported types: float16, float32, float64.
- **src_mask** (Tensor, optional) - The mask of the src sequence. The shape is (S, S) or $(N * nhead, S, S)$. Supported types: float16, float32, float64, bool. Default: `None`.
- **tgt_mask** (Tensor, optional) - The mask of the tgt sequence. The shape is (T, T) or $(N * nhead, T, T)$. Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_mask** (Tensor, optional) - The additive mask of the encoder output. The shape is (T, S) . Supported types: float16, float32, float64, bool. Default: `None`.
- **src_key_padding_mask** (Tensor, optional) - The mask of src keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.
- **tgt_key_padding_mask** (Tensor, optional) - The mask of tgt keys per batch. The shape is (T) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.
- **memory_key_padding_mask** (Tensor, optional) - The mask of memory keys per batch. The shape is (S) for unbatched input, otherwise (N, S) . Supported types: float16, float32, float64, bool. Default: `None`.

Outputs:

Tensor. The shape is (T, E) for unbatched input, otherwise if *batch_first=False* , the shape is (T, N, E) and if *batch_first=True* , the shape is (N, T, E).

Raises

- **ValueError** –If the batch sizes of the init argument *src* and *tgt* are not equal.
- **ValueError** –If the number of features of the init argument *src* and *tgt* is not equal to that of *d_model*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> transformer_model = ms.nn.Transformer(nhead=16, num_encoder_layers=12)
>>> src = ms.Tensor(np.random.rand(10, 32, 512), ms.float32)
>>> tgt = ms.Tensor(np.random.rand(20, 32, 512), ms.float32)
>>> out = transformer_model(src, tgt)
>>> print(out.shape)
(20, 32, 512)
```

3.7 Embedding Layer

API Name	Description	Supported Platforms
mindspore.nn.Embedding	A simple lookup table that stores embeddings of a fixed dictionary and size.	`Ascend` `GPU` `CPU`
mindspore.nn.EmbeddingLookup	EmbeddingLookup layer.	`Ascend` `GPU` `CPU`

continues on next page

Table 7 – continued from previous page

mindspore.nn.MultiFieldEmbeddingLookup	Returns a slice of input tensor based on the specified indices and the field ids.	Ascend GPU

3.7.1 mindspore.nn.Embedding

class mindspore.nn.**Embedding**(*vocab_size*, *embedding_size*, *use_one_hot=False*, *embedding_table='normal'*, *dtype=mstype.float32*, *padding_idx=None*)

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Note: When 'use_one_hot' is set to True, the type of the *x* must be mindspore.int32.

Parameters

- **vocab_size** (*int*) –Size of the dictionary of embeddings.
- **embedding_size** (*int*) –The size of each embedding vector.
- **use_one_hot** (*bool*) –Specifies whether to apply one_hot encoding form. Default: `False` .
- **embedding_table** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the embedding_table. Refer to class mindspore.common.initializer for the values of string when a string is specified. Default: `'normal'` .
- **dtype** (*mindspore.dtype*) –Data type of *x*. Default: `mstype.float32` .
- **padding_idx** (*int, None*) –When the padding_idx encounters index, the output embedding vector of this index will be initialized to zero. Default: `None` . The feature is inactivated.

Inputs:

- **x** (Tensor) - Tensor of shape (batch_size, x_length). The elements of the Tensor must be integer and not larger than vocab_size. Otherwise the corresponding embedding vector will be zero. The data type is int32 or int64.

Outputs:

Tensor of shape (batch_size, x_length, embedding_size).

Raises

- **TypeError** –If *vocab_size* or *embedding_size* is not an int.
- **TypeError** –If *use_one_hot* is not a bool.
- **ValueError** –If *padding_idx* is an int which not in range [0, *vocab_size*).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.Embedding(20000, 768,  True)
>>> x = Tensor(np.ones([8, 128]), mindspore.int32)
>>> # Maps the input word IDs to word embedding.
>>> output = net(x)
>>> result = output.shape
>>> print(result)
(8, 128, 768)
```

3.7.2 mindspore.nn.EmbeddingLookup

class mindspore.nn.**EmbeddingLookup**(*vocab_size*, *embedding_size*, *param_init='normal'*, *target='CPU'*, *slice_mode='batch_slice'*, *manual_shapes=None*, *max_norm=None*, *sparse=True*, *vocab_cache_size=0*, *dtype=mstype.float32*)

EmbeddingLookup layer. Same function as the embedding layer, mainly used for heterogeneous parallel scenarios where large-scale embedding layers exist when automatic parallelism or semi-automatic parallelism is present.

Note: When 'target' is set to 'CPU', this module will use P.EmbeddingLookup().set_device('CPU') which specified 'offset = 0' to lookup table. When 'target' is set to 'DEVICE', this module will use P.Gather() which specified 'axis = 0' to lookup table. In field slice mode, the manual_shapes must be given. It is a tuple ,where the element is vocab[i], vocab[i] is the row numbers for i-th part. This module does not support the PyNative mode.

Parameters

- **vocab_size** (*int*) –Size of the dictionary of embeddings.
- **embedding_size** (*int*) –The size of each embedding vector.
- **param_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the embedding_table. Refer to class *initializer* for the values of string when a string is specified. Default: `'normal'` .
- **target** (*str*) –Specifies the target where the op is executed. The value must in [`'DEVICE'` , `'CPU'`]. Default: `'CPU'` .
- **slice_mode** (*str*) –The slicing way in semi_auto_parallel/auto_parallel. Default: `'batch_slice'` .
  - batch_slice (str): Divides the input index tensor into batches and retrieves the corresponding embedding vectors. This is applicable when each sample has the same number of indices.
  - field_slice (str): Divides the input index tensor into fields and retrieves the corresponding embedding vectors. This is applicable when each sample may have a different number of indices, but have the same feature dimensions.
  - table_row_slice (str): Treats the input index tensor as a 2D table, divides it by rows, and retrieves the corresponding embedding vectors.
  - table_column_slice (str): Treats the input index tensor as a 2D table, divides it by columns, and retrieves the corresponding embedding vectors.
- **manual_shapes** (*tuple*) –The accompaniment array in field slice mode. Default: `None` .
- **max_norm** (*Union[float, None]*) –A maximum clipping value. The data type must be float16, float32 or None. Default: `None` .

- **sparse** (*bool*) –Using sparse mode. When 'target' is set to 'CPU', 'sparse' has to be true. Default: `True` .
- **vocab_cache_size** (*int*) –Cache size of the dictionary of embeddings. Default: `0` . It is valid only in parameter server trainning mode and 'DEVICE' target. And the moment parameter of corresponding optimizer will also be set to the cache size. In addition, it should be noted that it will cost the 'DEVICE' memory, so suggests setting a reasonable value to avoid insufficient memory.
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **input_indices** (Tensor) - The shape of tensor is $(y_1, y_2, ..., y_S)$. Specifies the indices of elements of the original Tensor. Values can be out of range of embedding_table, and the exceeding part will be filled with 0 in the output. Values does not support negative and the result is undefined if values are negative. Input_indices must only be a 2d tensor in this interface when run in semi auto parallel/auto parallel mode.

Outputs:

Tensor, the shape of tensor is $(z_1, z_2, ..., z_N)$.

Raises

- **TypeError** –If *vocab_size* or *embedding_size* or *vocab_cache_size* is not an int.
- **TypeError** –If *sparse* is not a bool or *manual_shapes* is not a tuple.
- **ValueError** –If *vocab_size* or *embedding_size* is less than 1.
- **ValueError** –If *vocab_cache_size* is less than 0.
- **ValueError** –If *target* is neither 'CPU' nor 'DEVICE'.
- **ValueError** –If *slice_mode* is not one of 'batch_slice' or 'field_slice' or 'table_row_slice' or 'table_column_slice'.
- **ValueError** –If *sparse* is False and *target* is 'CPU'.
- **ValueError** –If *slice_mode* is 'field_slice' and *manual_shapes* is None.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input_indices = Tensor(np.array([[1, 0], [3, 2]]), mindspore.int32)
>>> result = nn.EmbeddingLookup(4,2)(input_indices)
>>> print(result.shape)
(2, 2, 2)
```

3.7.3 mindspore.nn.MultiFieldEmbeddingLookup

class mindspore.nn.**MultiFieldEmbeddingLookup**(*vocab_size*, *embedding_size*, *field_size*, *param_init='normal'*, *target='CPU'*, *slice_mode='batch_slice'*, *feature_num_list=None*, *max_norm=None*, *sparse=True*, *operator='SUM'*, *dtype=mstype.float32*)

Returns a slice of input tensor based on the specified indices and the field ids. This operation supports looking up embeddings using multi hot and one hot fields simultaneously.

Note: When 'target' is set to 'CPU', this module will use P.EmbeddingLookup().set_device('CPU') which specified 'offset = 0' to lookup table. When 'target' is set to 'DEVICE', this module will use P.Gather() which specified 'axis = 0' to lookup table. The vectors with the same field_ids will be combined by the *operator*, such as 'SUM', 'MAX' and 'MEAN'. Ensure the input_values of the padded id is zero, so that they can be ignored. The final output will be zeros if the sum of absolute weight of the field is zero. This class only supports ['table_row_slice', 'batch_slice' and 'table_column_slice']. For the operation 'MAX' on device Ascend, there is a constraint where $batch\_size * (seq\_length + field\_size) < 3500$.

Parameters

- **vocab_size** (*int*) −The size of the dictionary of embeddings.
- **embedding_size** (*int*) −The size of each embedding vector.
- **field_size** (*int*) −The field size of the final outputs.
- **param_init** (*Union[Tensor, str, Initializer, numbers.Number]*) −Initializer for the embedding_table. Refer to class *initializer* for the values of string when a string is specified. Default: `'normal'` .
- **target** (*str*) −Specifies the target where the op is executed. The value must in [`'DEVICE'` , `'CPU'`]. Default: `'CPU'` .
- **slice_mode** (*str*) −The slicing way in semi_auto_parallel/auto_parallel. Default: `'batch_slice'`.
  - batch_slice (str): Divides the input index tensor into batches and retrieves the corresponding embedding vectors. This is applicable when each sample has the same number of indices.
  - field_slice (str): Divides the input index tensor into fields and retrieves the corresponding embedding vectors. This is applicable when each sample may have a different number of indices, but have the same feature dimensions.
  - table_row_slice (str): Treats the input index tensor as a 2D table, divides it by rows, and retrieves the corresponding embedding vectors.
  - table_column_slice (str): Treats the input index tensor as a 2D table, divides it by columns, and retrieves the corresponding embedding vectors.
- **feature_num_list** (*tuple*) −The accompaniment array in field slice mode. This is unused currently. Default: `None` .
- **max_norm** (*Union[float, None]*) −A maximum clipping value. The data type must be float16, float32. Default: `None` .
- **sparse** (*bool*) −Using sparse mode. When 'target' is set to `'CPU'` , 'sparse' has to be true. Default: `True` .
- **operator** (*str*) −The pooling method for the features in one field. Support `'SUM'` , `'MEAN'` and `'MAX'` . Default: `'SUM'` .
- **dtype** (*mindspore.dtype*) −Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **input_indices** (Tensor) - The shape of tensor is $(batch\_size, seq\_length)$. Specifies the indices of elements of the original Tensor. Input_indices must be a 2d tensor in this interface. Type is Int32, Int64.
- **input_values** (Tensor) - The shape of tensor is $(batch\_size, seq\_length)$. Specifies the weights of elements of the input_indices. The lookout vector will multiply with the input_values. Type is float32.
- **field_ids** (Tensor) - The shape of tensor is $(batch\_size, seq\_length)$. Specifies the field id of elements of the input_indices. Type is Int32.

Outputs:

Tensor, the shape of tensor is $(batch\_size, field\_size, embedding\_size)$. Type is float32.

Raises

- **TypeError** – If *vocab_size* or *embedding_size* or *field_size* is not an int.
- **TypeError** – If *sparse* is not a bool or *feature_num_list* is not a tuple.
- **ValueError** – If *vocab_size* or *embedding_size* or *field_size* is less than 1.
- **ValueError** – If *target* is neither `'CPU'` nor `'DEVICE'`.
- **ValueError** – If *slice_mode* is not one of `'batch_slice'`, `'field_slice'`, `'table_row_slice'`, `'table_column_slice'`.
- **ValueError** – If *sparse* is False and *target* is `'CPU'`.
- **ValueError** – If *slice_mode* is `'field_slice'` and *feature_num_list* is None.
- **ValueError** – If *operator* is not one of `'SUM'`, `'MAX'`, `'MEAN'`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> input_indices = Tensor([[2, 4, 6, 0, 0], [1, 3, 5, 0, 0]], mindspore.int32)
>>> input_values = Tensor([[1, 1, 1, 0, 0], [1, 1, 1, 0, 0]], mindspore.float32)
>>> field_ids = Tensor([[0, 1, 1, 0, 0], [0, 0, 1, 0, 0]], mindspore.int32)
>>> net = nn.MultiFieldEmbeddingLookup(10, 2, field_size=2, operator='SUM', target='DEVICE')
>>> out = net(input_indices, input_values, field_ids)
>>> print(out.shape)
(2, 2, 2)
```

3.8 Nonlinear Activation Layer

API Name	Description	Supported Platforms
mindspore.nn.CELU	CELU Activation Operator.	`Ascend` `GPU` `CPU`
mindspore.nn.ELU	Applies the exponential linear unit function element-wise.	`Ascend` `GPU` `CPU`
mindspore.nn.FastGelu	Applies FastGelu function to each element of the input.	`Ascend` `GPU` `CPU`

continues on next page

Table 8 – continued from previous page

mindspore.nn.GELU	Applies GELU function to each element of the input.	Ascend GPU CPU
mindspore.nn.GLU	The gated linear unit function.	Ascend GPU CPU
mindspore.nn.get_activation	Gets the activation function.	Ascend GPU CPU
mindspore.nn.Hardtanh	Applies the Hardtanh function element-wise.	Ascend GPU CPU
mindspore.nn.HShrink	Applies Hard Shrink activation function element-wise.	Ascend GPU CPU
mindspore.nn.HSigmoid	Applies Hard Sigmoid activation function element-wise.	Ascend GPU CPU
mindspore.nn.HSwish	Applies Hard Swish activation function element-wise.	Ascend GPU CPU
mindspore.nn.LeakyReLU	Leaky ReLU activation function.	Ascend GPU CPU
mindspore.nn.LogSigmoid	Applies logsigmoid activation element-wise.	Ascend GPU CPU
mindspore.nn.LogSoftmax	Applies the LogSoftmax function to n-dimensional input tensor element-wise.	Ascend GPU CPU
mindspore.nn.LRN	Local Response Normalization.	GPU CPU
mindspore.nn.Mish	Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.	Ascend GPU CPU
mindspore.nn.Softsign	Applies softsign activation function element-wise.	Ascend GPU CPU
mindspore.nn.PReLU	Applies PReLU activation function element-wise.	Ascend GPU CPU
mindspore.nn.ReLU	Applies ReLU (Rectified Linear Unit activation function) element-wise.	Ascend GPU CPU
mindspore.nn.ReLU6	Compute ReLU6 activation function element-wise.	Ascend GPU CPU
mindspore.nn.RReLU	Applies RReLU (Randomized Leaky ReLU activation function) element-wise.	Ascend GPU CPU
mindspore.nn.SeLU	Applies activation function SeLU (Scaled exponential Linear Unit) element-wise.	Ascend GPU CPU
mindspore.nn.SiLU	Applies the silu linear unit function element-wise.	Ascend GPU CPU
mindspore.nn.Sigmoid	Applies sigmoid activation function element-wise.	Ascend GPU CPU
mindspore.nn.Softmin	Softmin activation function, which is a two-category function mindspore.nn.Sigmoid in the promotion of multi-classification, and the purpose is to show the results of multi-classification in the form of probability.	Ascend GPU CPU
mindspore.nn.Softmax	Softmax activation function, which is a two-category function mindspore.nn.Sigmoid in the promotion of multi-classification, the purpose is to show the results of multi-classification in the form of probability.	Ascend GPU CPU
mindspore.nn.Softmax2d	Softmax function applied to 2D features data.	Ascend GPU CPU
mindspore.nn.SoftShrink	Applies the SoftShrink function element-wise.	Ascend GPU CPU
mindspore.nn.Tanh	Applies the Tanh function element-wise, returns a new tensor with the hyperbolic tangent of the elements of input, The input is a Tensor with any valid shape.	Ascend GPU CPU
mindspore.nn.Tanhshrink	Applies Tanhshrink activation function element-wise and returns a new tensor.	Ascend GPU CPU
mindspore.nn.Threshold	Thresholds each element of the input Tensor.	Ascend GPU CPU

3.8.1 mindspore.nn.CELU

class mindspore.nn.**CELU**(*alpha=1.0*)

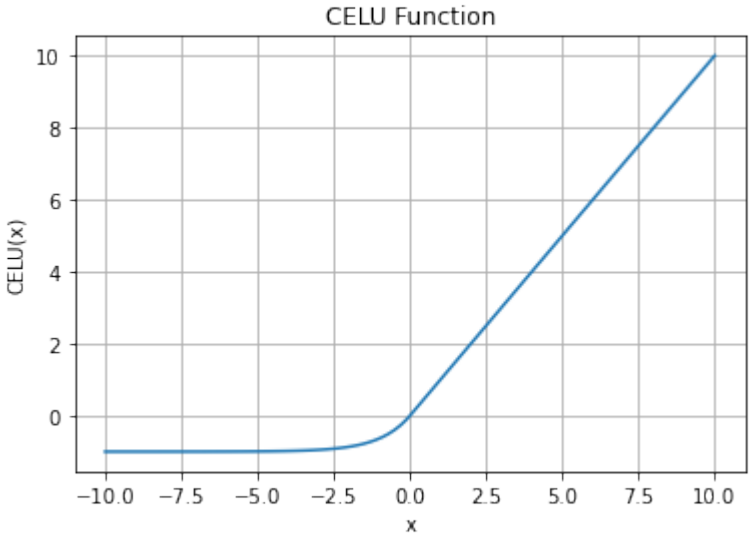
CELU Activation Operator.

Applies the continuously differentiable exponential linear units function element-wise.

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

For more details, refer to CELU .

CELU Activation Function Graph:



Parameters

alpha (*float, optional*) – The α value for the Celu formulation. Default: 1.0 .

Inputs:

- **x** (Tensor) - The input of CELU. The required dtype is float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

- **TypeError** – If *alpha* is not a float.
- **ValueError** – If *alpha* has the value of 0.
- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If the dtype of *x* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([-2.0, -1.0, 1.0, 2.0]), mindspore.float32)
>>> celu = nn.CELU()
>>> output = celu(x)
>>> print(output)
[-0.86466473 -0.63212055  1.          2.        ]
```

3.8.2 mindspore.nn.ELU

class mindspore.nn.**ELU**(*alpha=1.0*)

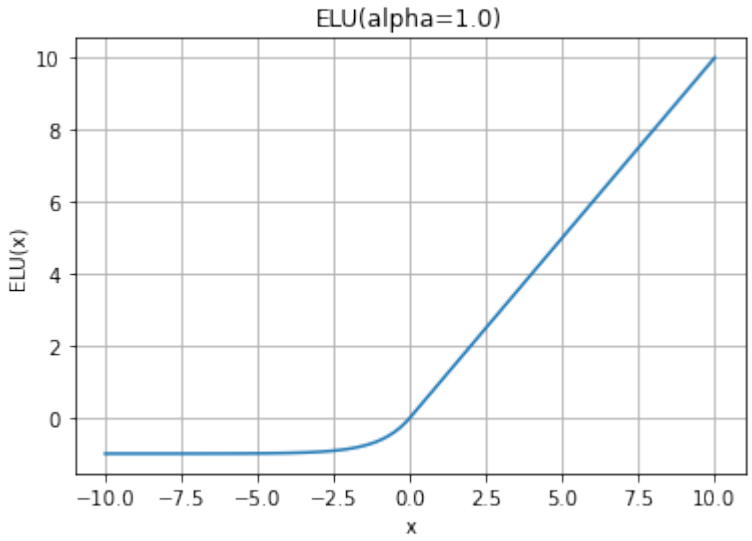
Applies the exponential linear unit function element-wise.

The activation function is defined as:

$$E_i = \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \alpha * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where x_i represents the element of the input and α represents the *alpha* parameter.

ELU Activation Function Graph:



Parameters

alpha (*float*) – The alpha value of ELU, the data type is float. Default: `1.0`. Only alpha equal to `1.0` is supported currently.

Inputs:

- **input_x** (Tensor) - The input of ELU is a Tensor of any dimension with data type of float16 or float32.

Outputs:

Tensor, with the same type and shape as the *input_x*.

Raises

- **TypeError** – If *alpha* is not a float.
- **TypeError** – If dtype of *input_x* is neither float16 nor float32.
- **ValueError** – If *alpha* is not equal to 1.0.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float32)
>>> elu = nn.ELU()
>>> result = elu(x)
>>> print(result)
[-0.63212055  -0.86466473  0.  2.  1.]
```

3.8.3 mindspore.nn.FastGelu

class mindspore.nn.**FastGelu**

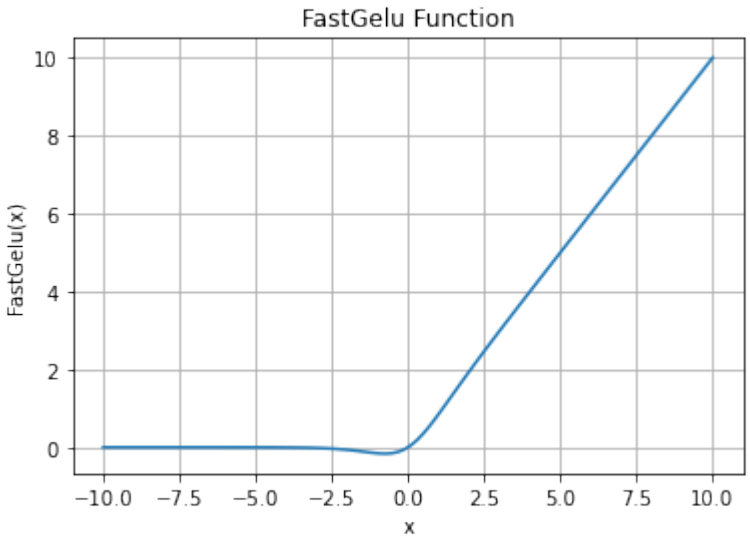
Applies FastGelu function to each element of the input. The input is a Tensor with any valid shape.

FastGelu is defined as:

$$FastGelu(x_i) = \frac{x_i}{1 + \exp(-1.702 * |x_i|)} * \exp(0.851 * (x_i - |x_i|))$$

where x_i is the element of the input.

FastGelu Activation Function Graph:



Inputs:

- **x** (Tensor) - The input of FastGelu with data type of float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

TypeError – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> fast_gelu = nn.FastGelu()
>>> output = fast_gelu(x)
>>> print(output)
[[-1.5418735e-01  3.9921875e+00 -9.7473649e-06]
 [ 1.9375000e+00 -1.0052517e-03  8.9824219e+00]]
```

3.8.4 mindspore.nn.GELU

class mindspore.nn.**GELU**(*approximate=True*)

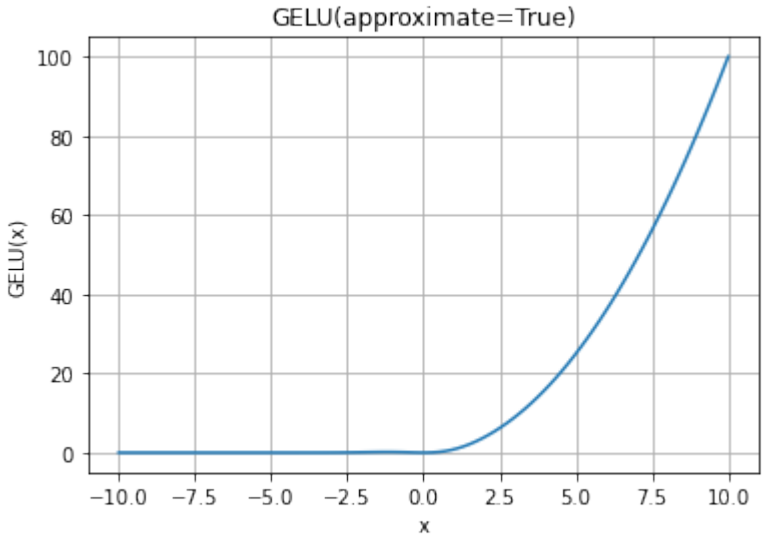
Applies GELU function to each element of the input. The input is a Tensor with any valid shape.

GELU is defined as:

$$GELU(x_i) = x_i * P(X < x_i),$$

where P is the cumulative distribution function of standard Gaussian distribution and x_i is the element of the input.

GELU Activation Function Graph:



Parameters

approximate (*bool, optional*) – Whether to enable approximation. Default: `True` .

If *approximate* is `True`, The gaussian error linear activation is:

$0.5 * x * (1 + tanh(\sqrt{(2/\pi)} * (x + 0.044715 * x^3)))$

else, it is:

$x * P(X <= x) = 0.5 * x * (1 + erf(x/\sqrt{(2)}))$, where $P(X)$ $N(0, 1)$.

Note:

- when calculating the input gradient of GELU with an input value of infinity, there are differences in the output of the backward between `Ascend` and `GPU`.
- when x is -inf, the computation result of `Ascend` is 0, and the computation result of `GPU` is Nan.
- when x is inf, the computation result of `Ascend` is dy, and the computation result of `GPU` is Nan.
- In mathematical terms, the result of Ascend has higher precision.

Inputs:

- **x** (Tensor) - The input of GELU with data type of float16, float32, or float64. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

TypeError – If dtype of *x* is not one of float16, float32, or float64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> gelu = nn.GELU()
>>> output = gelu(x)
>>> print(output)
[[-1.5880802e-01  3.9999299e+00 -3.1077917e-21]
 [ 1.9545976e+00 -2.2918017e-07  9.0000000e+00]]
>>> gelu = nn.GELU(approximate=False)
>>> # CPU not support "approximate=False", using "approximate=True" instead
>>> output = gelu(x)
>>> print(output)
[[-1.5865526e-01  3.9998732e+00 -0.0000000e+00]
 [ 1.9544997e+00 -1.4901161e-06  9.0000000e+00]]
```

3.8.5 mindspore.nn.GLU

class mindspore.nn.**GLU**(*axis=-1*)

The gated linear unit function.

$$GLU(a, b) = a \otimes \sigma(b)$$

where a is the first half of the input matrices and b is the second half.

Here σ is the sigmoid function, and $\otimes$ is the Hadamard product.

Parameters

axis (*int, optional*) – the axis to split the input. Default: `-1` , the last axis in x.

Inputs:

- **x** (Tensor) - $(*_1, N, *_2)$ where * means, any number of additional dimensions.

Outputs:

Tensor, the same dtype as the x, with the shape $(*_1, M, *_2)$ where $M = N/2$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> m = ms.nn.GLU()
>>> input = ms.Tensor([[0.1,0.2,0.3,0.4],[0.5,0.6,0.7,0.8]])
>>> output = m(input)
>>> print(output)
[[0.05744425 0.11973753]
 [0.33409387 0.41398472]]
```

3.8.6 mindspore.nn.get_activation

mindspore.nn.**get_activation**(*name*, *prim_name=None*)

Gets the activation function.

Parameters

- **name** (*str*) – The name of the activation function.
- **prim_name** (*Union[str, None]*) – The name of primitive. Default: `None` .

Returns

Function, the activation function.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>> sigmoid = nn.get_activation('sigmoid')
>>> print(sigmoid)
Sigmoid()
```

3.8.7 mindspore.nn.Hardtanh

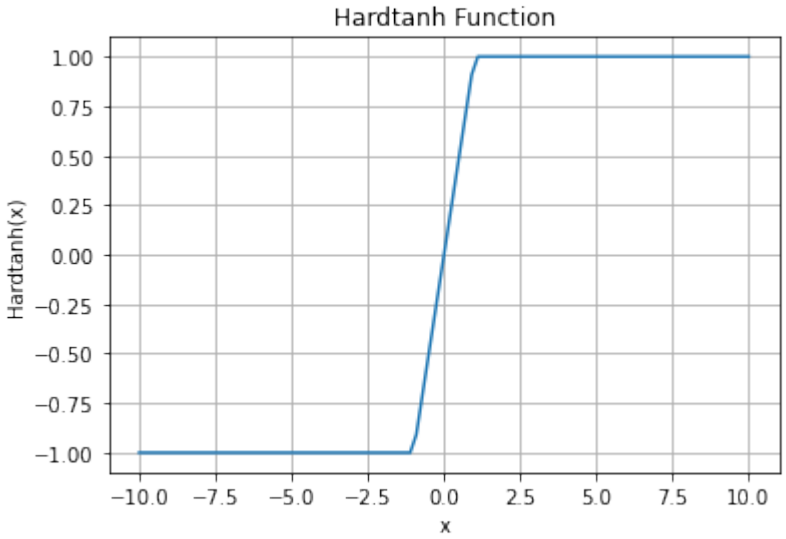
class mindspore.nn.**Hardtanh**(*min_val=- 1.0*, *max_val=1.0*)

Applies the Hardtanh function element-wise. The activation function is defined as:

$$\text{Hardtanh}(x) = \begin{cases} 1, & \text{if } x > 1; \\ -1, & \text{if } x < -1; \\ x, & \text{otherwise.} \end{cases}$$

Linear region range $[-1, 1]$ can be adjusted using *min_val* and *max_val*.

Hardtanh Activation Function Graph:



Note: On Ascend, data type of float16 might lead to accidental accuracy problem.

Parameters

- **min_val** (*Union[int, float]*) – Minimum value of the linear region range. Default: `-1.0`.
- **max_val** (*Union[int, float]*) – Maximum value of the linear region range. Default: `1.0`.

Inputs:

- **x** (Tensor) - Input Tensor with data type of float16 or float32. On CPU and Ascend support dimension 0-7D. On GPU support dimension 0-4D.

Outputs:

Tensor, with the same dtype and shape as *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **TypeError** – If dtype of *min_val* is neither float nor int.
- **TypeError** – If dtype of *max_val* is neither float nor int.
- **ValueError** – If *min_val* is not less than *max_val*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> hardtanh = nn.Hardtanh(min_val=-1.0, max_val=1.0)
>>> output = hardtanh(x)
>>> print(output)
[-1. -1.  0.  1.  1.]
```

3.8.8 mindspore.nn.HShrink

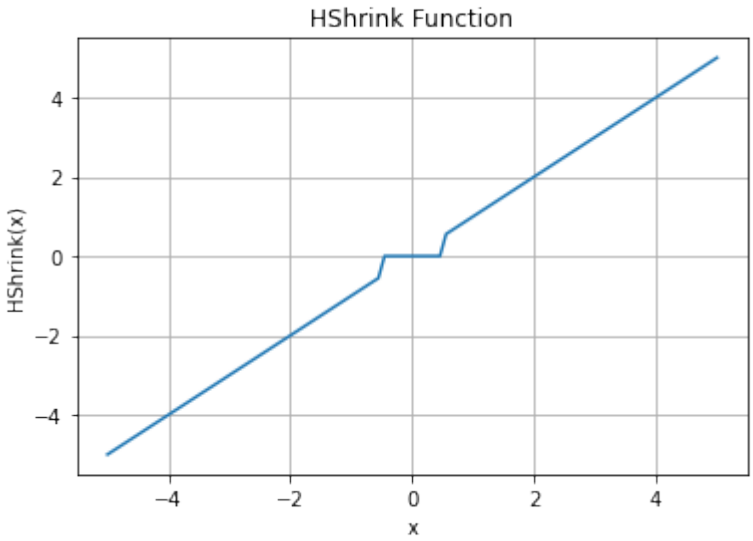
class mindspore.nn.**HShrink**(*lambd=0.5*)

Applies Hard Shrink activation function element-wise.

The formula is defined as follows:

$$\text{HShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

HShrink Activation Function Graph:



Parameters

lambd (*number, optional*) – The threshold λ defined by the Hard Shrink formula. Default: `0.5`.

Inputs:

- **input** (Tensor) - The input of Hard Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32.

Outputs:

Tensor, the same shape and data type as the input.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([[0.5, 1, 2.0], [0.0533, 0.0776, -2.1233]]), mindspore.float32)
>>> hshrink = nn.HShrink()
>>> output = hshrink(input)
>>> print(output)
[[ 0.      1.      2.    ]
 [ 0.      0.     -2.1233]]
```

3.8.9 mindspore.nn.HSigmoid

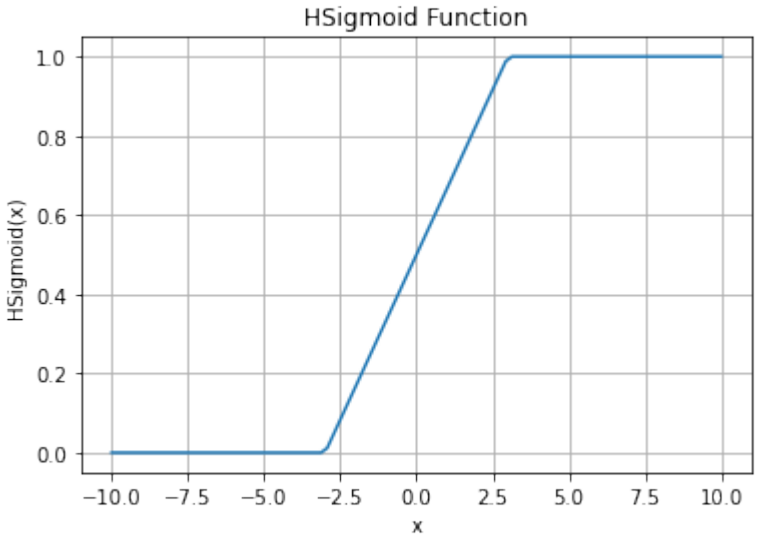
class mindspore.nn.**HSigmoid**

Applies Hard Sigmoid activation function element-wise.

Hard Sigmoid is defined as:

$$\text{HSigmoid}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ 1, & \text{if } input \geq +3, \\ input/6 + 1/2, & \text{otherwise} \end{cases}$$

HSigmoid Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of HSigmoid.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> hsigmoid = nn.HSigmoid()
>>> result = hsigmoid(input)
>>> print(result)
[0.3333 0.1666 0.5    0.8335 0.6665]
```

3.8.10 mindspore.nn.HSwish

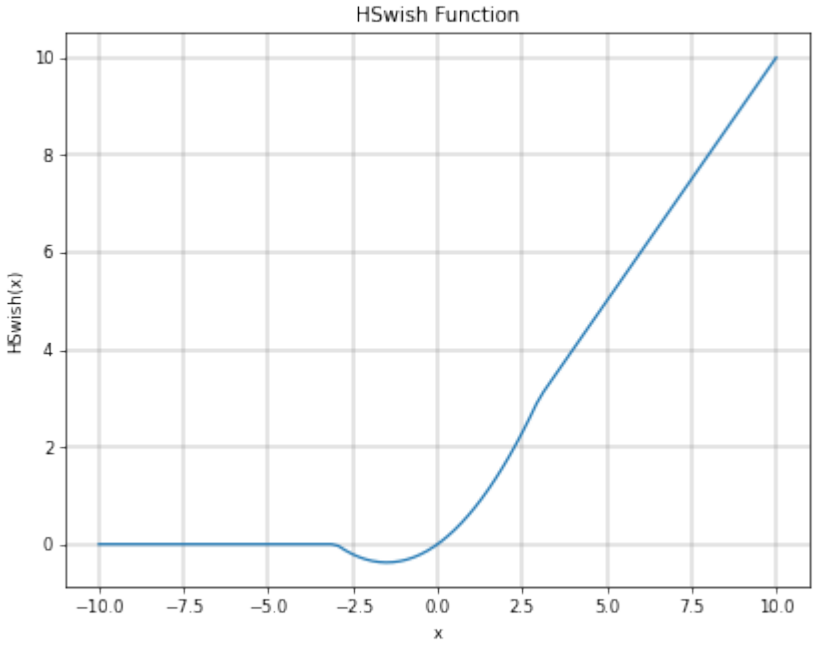
class mindspore.nn.**HSwish**

Applies Hard Swish activation function element-wise.

Hard swish is defined as:

$$\text{HSwish}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ input, & \text{if } input \geq +3, \\ input * (input + 3)/6, & \text{otherwise} \end{cases}$$

HSwish Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of HSwish.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> hswish = nn.HSwish()
>>> result = hswish(input)
>>> print(result)
[-0.3333 -0.3333  0.      1.667   0.6665]
```

3.8.11 mindspore.nn.LeakyReLU

class mindspore.nn.**LeakyReLU**(*alpha=0.2*)

Leaky ReLU activation function.

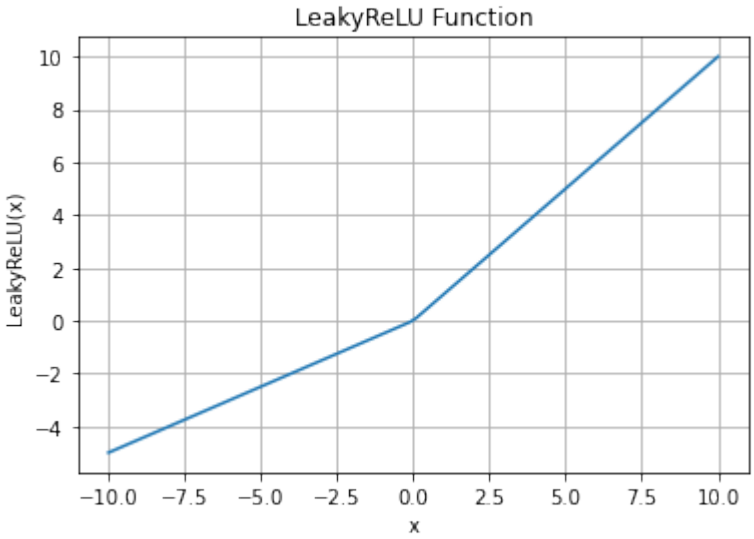
The activation function is defined as:

$$\text{leaky\_relu}(x) = \begin{cases} x, & \text{if } x \geq 0; \\ \alpha * x, & \text{otherwise.} \end{cases}$$

where α represents the *alpha* parameter.

For more details, see Rectifier Nonlinearities Improve Neural Network Acoustic Models.

LeakyReLU Activation Function Graph:



Parameters

alpha (*Union[int, float]*) – Slope of the activation function at x < 0. Default: `0.2` .

Inputs:

- **x** (Tensor) - The input of LeakyReLU is a Tensor of any dimension.

Outputs:

Tensor, has the same type and shape as the *x*.

Raises

TypeError – If *alpha* is not a float or an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> leaky_relu = nn.LeakyReLU()
>>> output = leaky_relu(x)
>>> print(output)
[[-0.2  4.  -1.6]
 [ 2.  -1.   9. ]]
```

3.8.12 mindspore.nn.LogSigmoid

class mindspore.nn.**LogSigmoid**

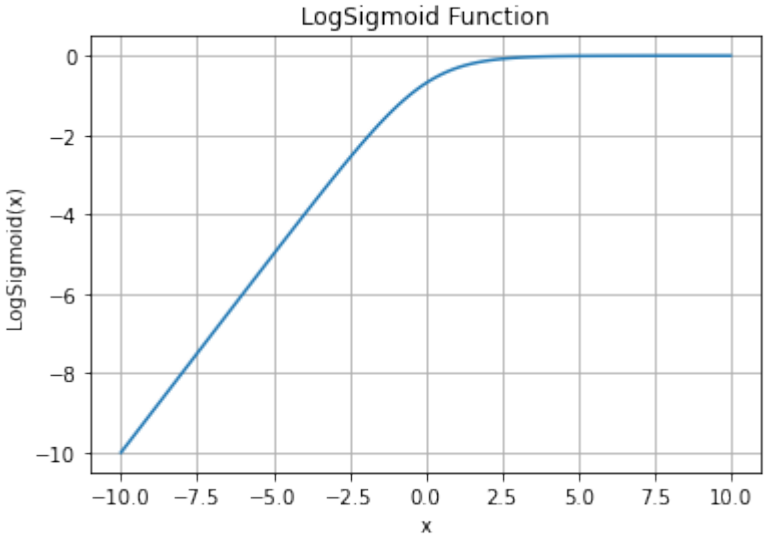
Applies logsigmoid activation element-wise. The input is a Tensor with any valid shape.

Logsigmoid is defined as:

$$\text{logsigmoid}(x_i) = \log(\frac{1}{1 + \exp(-x_i)}),$$

where x_i is the element of the input.

LogSigmoid Activation Function Graph:



Inputs:

- **x** (Tensor) - The input of LogSigmoid with data type of float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

TypeError – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> net = nn.LogSigmoid()
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> output = net(x)
>>> print(output)
[-0.31326166 -0.12692806 -0.04858734]
```

3.8.13 mindspore.nn.LogSoftmax

class mindspore.nn.**LogSoftmax**(*axis=-1*)

Applies the LogSoftmax function to n-dimensional input tensor element-wise.

The input is transformed by the Softmax function and then by the log function to lie in range[-inf,0).

Logsoftmax is defined as:

$$\text{logsoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_{j=0}^{n-1}\exp(x_j)}\right)$$

Parameters

axis (*int*) – The axis to apply LogSoftmax operation, -1 means the last dimension. Default: `-1` .

Inputs:

- **x** (Tensor) - The input of LogSoftmax, with float16 or float32 data type.

Outputs:

Tensor, which has the same type and shape as *x* with output values in the range[-inf,0).

Raises

- **TypeError** – If *axis* is not an int.
- **TypeError** – If dtype of *x* is neither float16 nor float32.
- **ValueError** – If *axis* is not in range [-len(x), len(x)).

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> log_softmax = nn.LogSoftmax()
>>> output = log_softmax(x)
>>> print(output)
[[-5.00672150e+00 -6.72150636e-03 -1.20067215e+01]
 [-7.00091219e+00 -1.40009127e+01 -9.12250078e-04]]
```

3.8.14 mindspore.nn.LRN

class mindspore.nn.**LRN**(*depth_radius=5*, *bias=1.0*, *alpha=1.0*, *beta=0.5*, *norm_region='ACROSS_CHANNELS'*)

Local Response Normalization.

> **Warning:** LRN is deprecated on Ascend due to potential accuracy problem. It's recommended to use other normalization methods, e.g. *mindspore.nn.BatchNorm1d* , *mindspore.nn.BatchNorm2d* , *mindspore.nn.BatchNorm3d*.

Refer to *mindspore.ops.lrn()* for more details.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input_x = Tensor(np.array([[[[0.1], [0.2]],
...                            [[0.3], [0.4]]]]), mindspore.float32)
>>> output = nn.LRN()(input_x)
>>> print(output)
[[[[0.09534626]
   [0.1825742 ]]
  [[0.2860388 ]
   [0.3651484 ]]]]
```

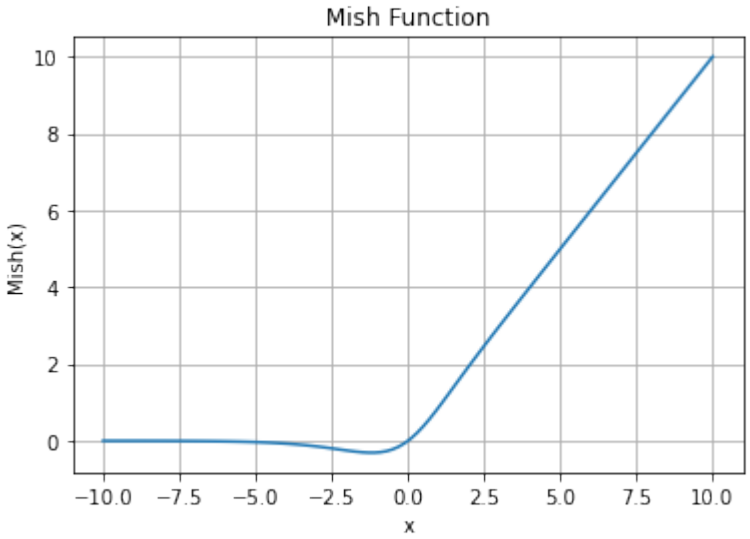
3.8.15 mindspore.nn.Mish

class mindspore.nn.**Mish**

Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.

Refer to *mindspore.ops.mish()* for more details.

Mish Activation Function Graph:



Supported Platforms:

Ascend GPU CPU

Examples

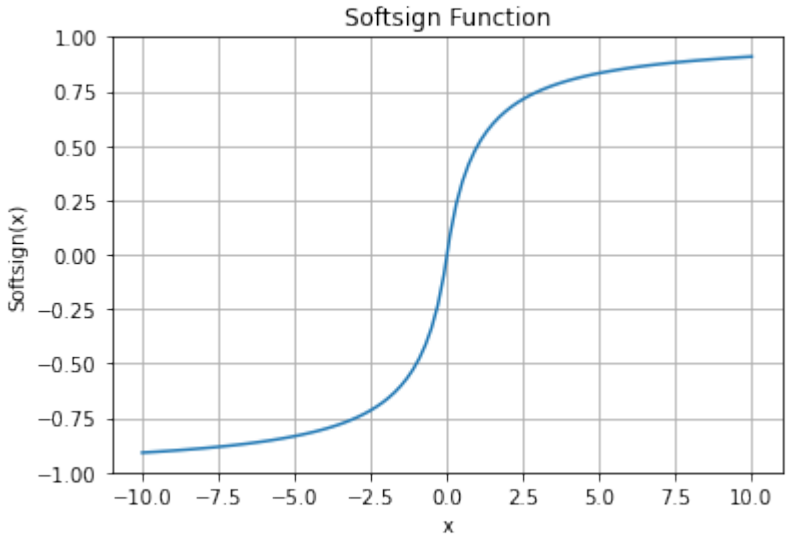
```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> mish = nn.Mish()
>>> output = mish(x)
>>> print(output)
[[-3.03401530e-01 3.99741292e+00 -2.68321624e-03]
 [ 1.94395900e+00 -3.35762873e-02 9.00000000e+00]]
```

3.8.16 mindspore.nn.Softsign

class mindspore.nn.**Softsign**

Applies softsign activation function element-wise.

Softsign Activation Function Graph:



Refer to *mindspore.ops.softsign()* for more details.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([0, -1, 2, 30, -30]), mindspore.float32)
>>> softsign = nn.Softsign()
>>> output = softsign(x)
>>> print(output)
[ 0.        -0.5         0.6666667  0.9677419 -0.9677419]
```

3.8.17 mindspore.nn.PReLU

class mindspore.nn.**PReLU**(*channel=1*, *w=0.25*)

Applies PReLU activation function element-wise.

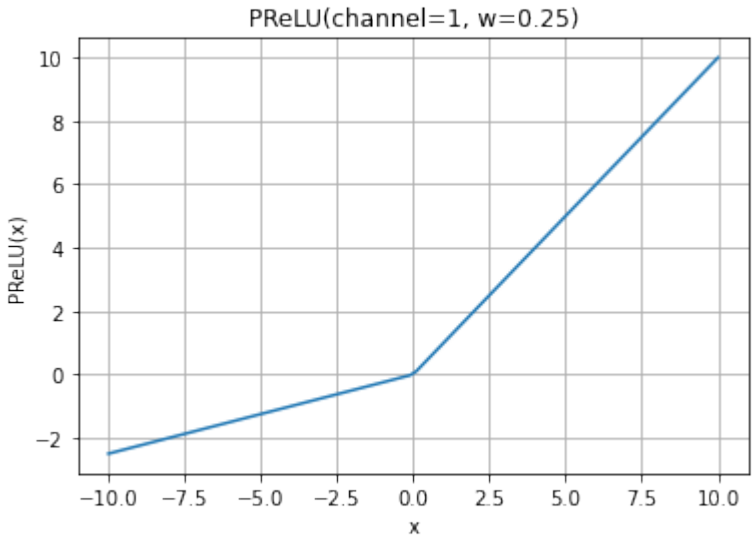
PReLU is defined as:

$$PReLU(x_i) = \max(0, x_i) + w * \min(0, x_i),$$

where x_i is an element of an channel of the input.

Here w is a learnable parameter with a default initial value `0.25`. Parameter w has dimensionality of the argument channel. If called without argument channel, a single parameter w will be shared across all channels.

PReLU Activation Function Graph:



Parameters

- **channel** (*int, optional*) – The elements number of parameter w. It could be an int, and the value is `1` or the channels number of input tensor x. Default: `1` .
- **w** (*Union[float, list, Tensor], optional*) – The initial value of parameter. It could be a float, a float list or a tensor has the same dtype as the input tensor x. Default: `0.25` .

Inputs:

- **x** (Tensor) - The input of PReLU with data type of float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same dtype and shape as the x.

Raises

- **TypeError** – If *channel* is not an int.
- **TypeError** – If w is not one of a float, a list[float], a Tensor[float].
- **TypeError** – If dtype of x is neither float16 nor float32.
- **ValueError** – If the x is a 0-D or 1-D Tensor on Ascend.

- **ValueError** – If *channel* is less than 1.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[[[0.1, 0.6], [0.9, 0.9]]]]), mindspore.float32)
>>> prelu = nn.PReLU()
>>> output = prelu(x)
>>> print(output)
[[[[0.1 0.6]
   [0.9 0.9]]]]
```

3.8.18 mindspore.nn.ReLU

class mindspore.nn.**ReLU**

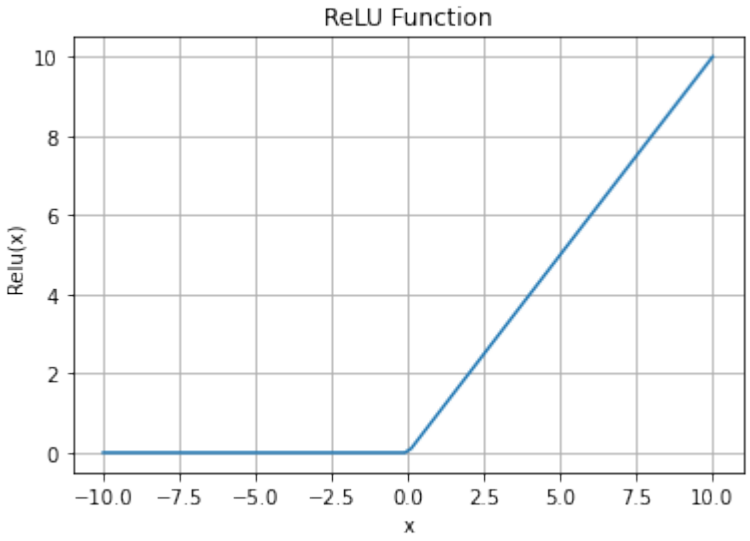
Applies ReLU (Rectified Linear Unit activation function) element-wise.

$$\text{ReLU}(input) = (input)^+ = \max(0, input),$$

It returns element-wise $\max(0, input)$.

Note: The neurons with the negative output will be suppressed and the active neurons will stay the same.

ReLU Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of ReLU is a Tensor of any dimension.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is not supported.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> relu = nn.ReLU()
>>> output = relu(input)
>>> print(output)
[0. 2. 0. 2. 0.]
```

3.8.19 mindspore.nn.ReLU6

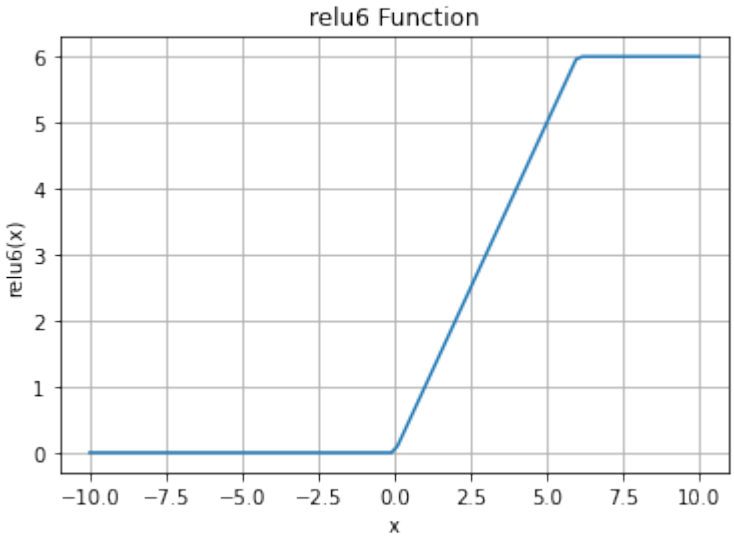
class mindspore.nn.**ReLU6**

Compute ReLU6 activation function element-wise.

ReLU6 is similar to ReLU with a upper limit of 6, which if the inputs are greater than 6, the outputs will be suppressed to 6. It computes element-wise as

$$Y = \min(\max(0, x), 6)$$

ReLU6 Activation Function Graph:



Inputs:

- **x** (Tensor) - The input of ReLU6 with data type of float16 or float32 and that is a Tensor of any valid shape.

Outputs:

Tensor, which has the same type as *x*.

Raises

TypeError –If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> relu6 = nn.ReLU6()
>>> output = relu6(x)
>>> print(output)
[0. 0. 0. 2. 1.]
```

3.8.20 mindspore.nn.RReLU

class mindspore.nn.**RReLU**(*lower=1 / 8*, *upper=1 / 3*)

Applies RReLU (Randomized Leaky ReLU activation function) element-wise.

The activation function is defined as:

$$\text{RReLU}(x_{ji}) = \begin{cases} x_{ji}, & \text{if } x_{ji} \geq 0; \\ \alpha_{ji} * x_{ji}, & \text{otherwise.} \end{cases}$$

where $\alpha_{ji} \sim U(l, u), l \leq u$.

Applies the RReLU function elementally, as described in the paper: Empirical Evaluation of Rectified Activations in Convolution Network .

Parameters

- **lower** (*Union[int, float]*) –Slope of the activation function at x < 0. Default: `1 / 8` .
- **upper** (*Union[int, float]*) –Slope of the activation function at x < 0. Default: `1 / 3` .

Inputs:

- **x** (Tensor) - The input of RReLU is a Tensor of any dimension.

Outputs:

Tensor, after RReLU, has the same type and shape as the *x*.

Raises

- **TypeError** –If *lower* is not a float or an int.
- **TypeError** –If *upper* is not a float or an int.

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *x* is not a Tensor of mindspore.float16 or mindspore.float32.
- **ValueError** – If *lower* is greater than upper.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

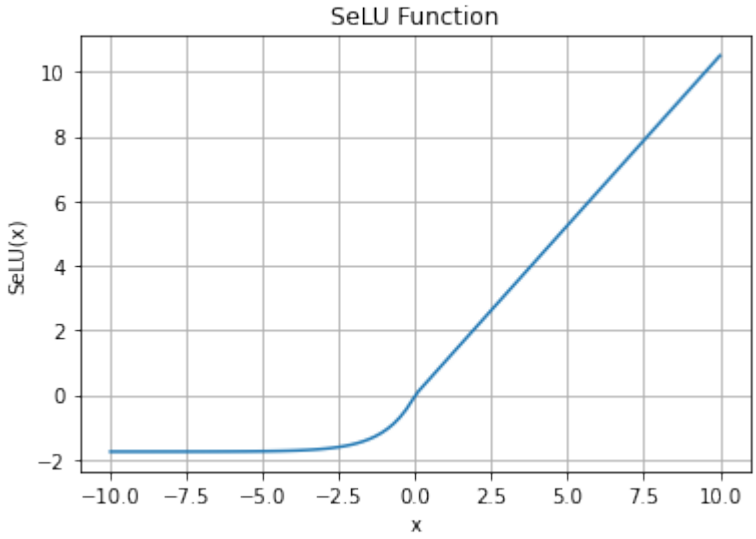
```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0], [2.0, 0]]), mindspore.float32)
>>> r_relu = nn.RReLU()
>>> output = r_relu(x)
>>> print(output)
[[-0.31465699  4.        ]
 [ 2.          0.        ]]
```

3.8.21 mindspore.nn.SeLU

class mindspore.nn.**SeLU**

Applies activation function SeLU (Scaled exponential Linear Unit) element-wise.

SeLU Activation Function Graph:



Refer to *mindspore.ops.selu()* for more details.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input_x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> selu = nn.SeLU()
>>> output = selu(input_x)
>>> print(output)
[[-1.1113307 4.202804 -1.7575096]
 [ 2.101402 -1.7462534 9.456309 ]]
```

3.8.22 mindspore.nn.SiLU

class mindspore.nn.**SiLU**

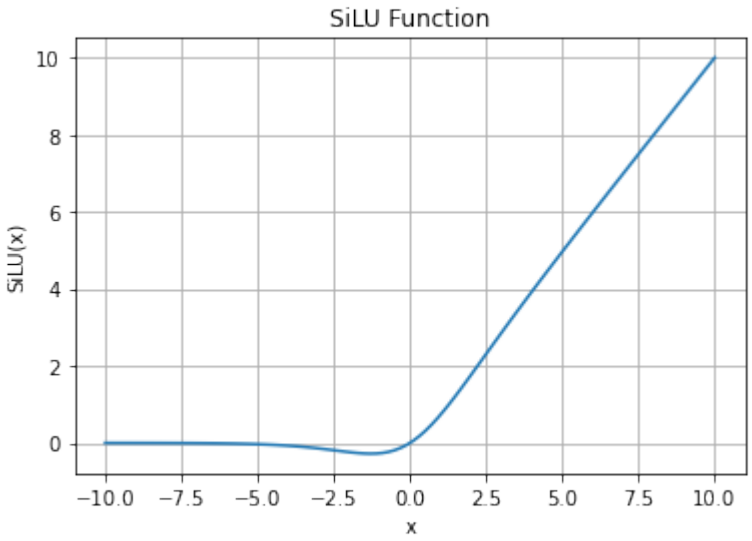
Applies the silu linear unit function element-wise.

$$\text{SiLU}(x) = x * \sigma(x),$$

where x_i is an element of the input, $\sigma(x)$ is Sigmoid function.

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

SiLU Activation Function Graph:



Inputs:

- **input** (Tensor) - *input* is x in the preceding formula. Input with the data type float16 or float32. Tensor of any dimension.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> silu = nn.SiLU()
>>> output = silu(input)
>>> print(output)
[-0.269  1.762  -0.1423  1.762  -0.269]
```

3.8.23 mindspore.nn.Sigmoid

class mindspore.nn.**Sigmoid**

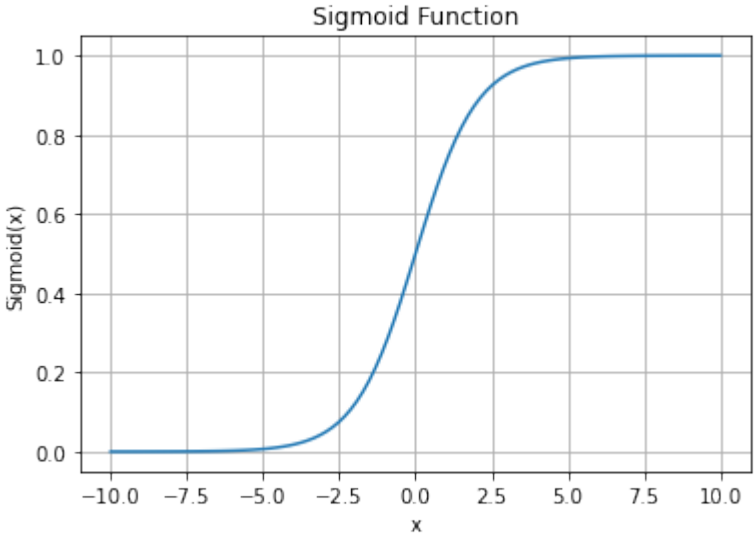
Applies sigmoid activation function element-wise.

Sigmoid function is defined as:

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

where x_i is the element of x.

Sigmoid Activation Function Graph:



Inputs:

- **input** (Tensor) - *input* is x in the preceding formula. Tensor of any dimension, the data type is float16, float32, float64, complex64 or complex128.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** –If dtype of *input* is not float16, float32, float64, complex64 or complex128.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> sigmoid = nn.Sigmoid()
>>> output = sigmoid(input)
>>> print(output)
[0.2688  0.11914 0.5     0.881   0.7305 ]
```

3.8.24 mindspore.nn.Softmin

class mindspore.nn.**Softmin**(*axis=-1*)

Softmin activation function, which is a two-category function *mindspore.nn.Sigmoid* in the promotion of multi-classification, and the purpose is to show the results of multi-classification in the form of probability.

Calculate the value of the exponential function for the elements of the input Tensor on the *axis*, and then normalized to lie in range [0, 1] and sum up to 1.

Softmin is defined as:

$$\text{softmin}(x_i) = \frac{\exp(-x_i)}{\sum_{j=0}^{n-1}\exp(-x_j)},$$

where x_i is the i-th slice in the given dimension of the input Tensor.

Parameters

axis (*Union[int, tuple[int]], optional*) –The axis to apply Softmin operation, if the dimension of input x is x.ndim, the range of axis is $[-x.ndim, x.ndim)$. -1 means the last dimension. Default: `-1` . In CPU environment, *axis* only supports int type.

Inputs:

- **x** (Tensor) - Tensor for computing Softmin functions with data type of float16 or float32.

Outputs:

Tensor, which has the same type and shape as x with values in the range $[0, 1]$.

Raises

- **TypeError** –If *axis* is neither an int nor a tuple.
- **TypeError** –If dtype of x is neither float16 nor float32.
- **ValueError** –If *axis* is a tuple whose length is less than 1.
- **ValueError** –If *axis* is a tuple whose elements are not all in the range $[-x.ndim, x.ndim)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # axis = -1(default), and the sum of return value is 1.0.
>>> x = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> softmin = nn.Softmin()
>>> output = softmin(x)
>>> print(output)
[0.2341  0.636  0.0862  0.01165  0.03168 ]
```

3.8.25 mindspore.nn.Softmax

class mindspore.nn.**Softmax**(*axis=-1*)

Softmax activation function, which is a two-category function *mindspore.nn.Sigmoid* in the promotion of multi-classification, the purpose is to show the results of multi-classification in the form of probability.

Calculate the value of the exponential function for the elements of the input Tensor on the *axis*, and then normalized to lie in range [0, 1] and sum up to 1.

Softmax is defined as:

$$\text{softmax}(input_i) = \frac{\exp(input_i)}{\sum_{j=0}^{n-1}\exp(input_j)},$$

where $input_i$ is the i-th slice in the given dimension of the input Tensor.

Parameters

axis (*int, optional*) – The axis to apply Softmax operation, if the dimension of *input* is input.ndim, the range of axis is *[-input.ndim, input.ndim)*, -1 means the last dimension. Default: `-1` .

Inputs:

- **input** (Tensor) - The input of Softmax.

Outputs:

Tensor, which has the same type and shape as *input* with values in the range[0, 1].

Raises

- **TypeError** – If *axis* is neither an int nor a tuple.
- **ValueError** – If *axis* is a tuple whose length is less than 1.
- **ValueError** – If *axis* is a tuple whose elements are not all in range *[-input.ndim, input.ndim)*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # axis = -1(default), and the sum of return value is 1.0.
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> softmax = nn.Softmax()
>>> output = softmax(input)
>>> print(output)
[0.03168 0.01166 0.0861  0.636   0.2341 ]
```

3.8.26 mindspore.nn.Softmax2d

class mindspore.nn.**Softmax2d**

Softmax function applied to 2D features data.

Applies *Softmax* to each location with an input Tensor of shape (C, H, W) .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}). The input of Softmax with data type of float16 or float32.

Outputs:

Tensor, which has the same type and shape as x with values in the range[0,1].

Raises

- **TypeError** – If dtype of x is neither float16 nor float32.
- **ValueError** – If *data_format* is neither 'NCHW' nor 'CHW'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[[[0.1, 0.2]], [[0.3, 0.4]], [[0.6, 0.5]]]]), mindspore.float32)
>>> softmax2d = nn.Softmax2d()
>>> output = softmax2d(x)
>>> print(output)
[[[[0.25838965 0.28001308]]
[[0.31559783 0.34200877]]
[[0.42601252 0.37797815]]]]
```

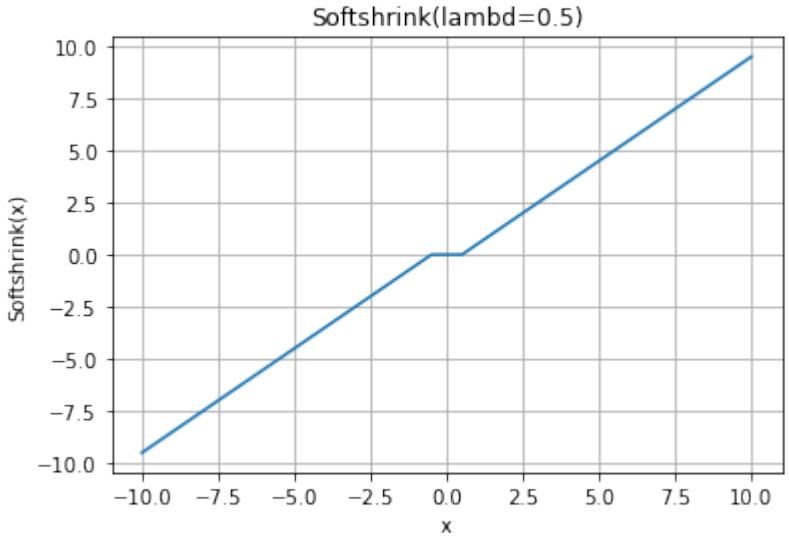
3.8.27 mindspore.nn.SoftShrink

class mindspore.nn.**SoftShrink**(*lambd=0.5*)

Applies the SoftShrink function element-wise.

$$\text{SoftShrink}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

SoftShrink Activation Function Graph:



Parameters

lambd (*number, optional*) – The threshold λ defined by the Soft Shrink formula. It should be greater than or equal to 0, default: 0.5.

Inputs:

- **input** (Tensor) - The input of Soft Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32.

Outputs:

Tensor, the same shape and data type as the input.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([[ 0.5297,  0.7871,  1.1754], [ 0.7836,  0.6218, -1.1542]]),␣
↪mindspore.float16)
>>> softshrink = nn.SoftShrink()
>>> output = softshrink(input)
>>> print(output)
[[ 0.02979  0.287    0.676  ]
 [ 0.2837   0.1216  -0.6543 ]]
```

3.8.28 mindspore.nn.Tanh

class mindspore.nn.**Tanh**

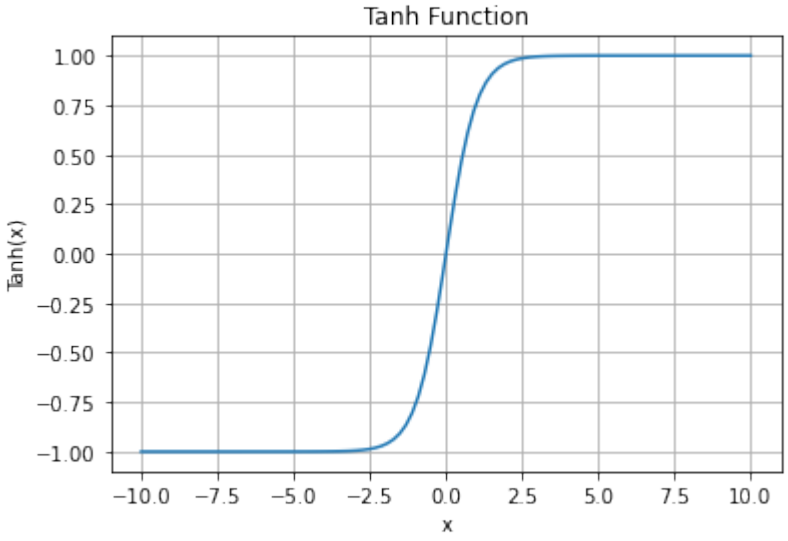
Applies the Tanh function element-wise, returns a new tensor with the hyperbolic tangent of the elements of input, The input is a Tensor with any valid shape.

Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Tanh Activation Function Graph:



Inputs:

- **x** (Tensor) - Tensor of any dimension, input with data type of float16 or float32.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

TypeError – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([1, 2, 3, 2, 1]), mindspore.float16)
>>> tanh = nn.Tanh()
>>> output = tanh(x)
>>> print(output)
[0.7617 0.964  0.995  0.964  0.7617]
```

3.8.29 mindspore.nn.Tanhshrink

class mindspore.nn.**Tanhshrink**

Applies Tanhshrink activation function element-wise and returns a new tensor.

Tanh function is defined as:

$$tanhshrink(x_i) = x_i - \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = x_i - \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Inputs:

- **x** (Tensor) - Tensor of any dimension.

Outputs:

Tensor, with the same shape as the x.

Raises

TypeError – If x is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([1, 2, 3, 2, 1]), ms.float16)
>>> tanhshrink = nn.Tanhshrink()
>>> output = tanhshrink(x)
>>> print(output)
[0.2383 1.036  2.004  1.036  0.2383]
```

3.8.30 mindspore.nn.Threshold

class mindspore.nn.**Threshold**(*threshold*, *value*)

Thresholds each element of the input Tensor.

The formula is defined as follows:

$$y = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

Parameters

- **threshold** (*Union[int, float]*) – The value to threshold at.
- **value** (*Union[int, float]*) – The value to replace with when element is less than threshold.

Inputs:

- **input_x** (Tensor) - The input of Threshold with data type of float16 or float32.

Outputs:

Tensor, the same shape and data type as the *input_x*.

Raises

- **TypeError** – If *threshold* is not a float or an int.
- **TypeError** – If *value* is not a float or an int.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> m = nn.Threshold(0.1, 20)
>>> inputs = Tensor([0.1, 0.2, 0.3], mindspore.float32)
>>> outputs = m(inputs)
>>> print(outputs)
[ 20.0     0.2      0.3]
```

3.9 Linear Layer

API Name	Description	Supported Platforms
mindspore.nn.Dense	The dense connected layer.	`Ascend GPU CPU`
mindspore.nn.BiDense	The bilinear dense connected layer.	`Ascend GPU CPU`

3.9.1 mindspore.nn.Dense

class mindspore.nn.**Dense**(*in_channels*, *out_channels*, *weight_init=None*, *bias_init=None*, *has_bias=True*, *activation=None*, *dtype=mstype.float32*)

The dense connected layer.

Applies dense connected layer for the input. This layer implements the operation as:

$$\text{outputs} = \text{activation}(X * \text{kernel} + \text{bias}),$$

where X is the input tensors, activation is the activation function passed as the activation argument (if passed in), kernel is a weight matrix with the same data type as the X created by the layer, and bias is a bias vector with the same data type as the X created by the layer (only if *has_bias* is `True`).

Warning: On the Ascend platform, if *bias* is `False` , the *x* cannot be greater than 6D in PYNATIVE or KBK mode.

Parameters

- **in_channels** (*int*) –The number of channels in the input space.
- **out_channels** (*int*) –The number of channels in the output space.
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable weight_init parameter. The dtype is same as *x*. The values of str refer to the function *initializer*. Default: `None` , weight will be initialized using HeUniform.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable bias_init parameter. The dtype is same as *x*. The values of str refer to the function *initializer*. Default: `None` , bias will be initialized using Uniform.
- **has_bias** (*bool, optional*) –Specifies whether the layer uses a bias vector bias. Default: `True`.
- **activation** (*Union[str, Cell, Primitive, None], optional*) –activate function applied to the output of the fully connected layer. Both activation name, e.g. 'relu', and mindspore activation function, e.g. mindspore.ops.ReLU(), are supported. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) –Data type of Parameter. Default: `mstype.float32` . When *weight_init* is Tensor, Parameter has the same data type as *weight_init* , in other cases, Parameter has the same data type as *dtype*, the same goes for *bias_init*.

Inputs:

- **x** (Tensor) - Tensor of shape $(*, in\_channels)$. The *in_channels* in *Args* should be equal to $in\_channels$ in *Inputs*.

Outputs:

Tensor of shape $(*, out\_channels)$.

Raises

- **TypeError** –If *in_channels* or *out_channels* is not an int.
- **TypeError** –If *has_bias* is not a bool.
- **TypeError** –If *activation* is not one of str, Cell, Primitive, None.
- **ValueError** –If length of shape of *weight_init* is not equal to 2 or shape[0] of *weight_init* is not equal to *out_channels* or shape[1] of *weight_init* is not equal to *in_channels*.
- **ValueError** –If length of shape of *bias_init* is not equal to 1 or shape[0] of *bias_init* is not equal to *out_channels*.

- **RuntimeError** –On the Ascend platform, if *bias* is `False` and *x* is greater than 6D in PYNATIVE or KBK mode.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[180, 234, 154], [244, 48, 247]]), mindspore.float32)
>>> net = nn.Dense(3, 4)
>>> output = net(x)
>>> print(output.shape)
(2, 4)
```

3.9.2 mindspore.nn.BiDense

class mindspore.nn.**BiDense**(*in1_channels*, *in2_channels*, *out_channels*, *weight_init=None*, *bias_init=None*, *has_bias=True*, *dtype=mstype.float32*)

The bilinear dense connected layer.

Applies dense connected layer for two inputs. This layer implements the operation as:

$$y = x_1^T A x_2 + b,$$

where x_1 is the first input tensor, x_2 is the second input tensor , A is a weight matrix with the same data type as the x_* created by the layer , and b is a bias vector with the same data type as the x_* created by the layer (only if has_bias is `True`).

Parameters

- **in1_channels** (*int*) –The number of channels in the input1 space.
- **in2_channels** (*int*) –The number of channels in the input2 space.
- **out_channels** (*int*) –The number of channels in the output space.
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –The trainable weight_init parameter. The values of str refer to the function *initializer*. Default: `None` .
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –The trainable bias_init parameter. The values of str refer to the function *initializer*. Default: `None` .
- **has_bias** (*bool*) –Specifies whether the layer uses bias vector. Default: `True` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Shape:

- **input1** - $(*, H_{in1})$ where H_{in1} = in1_channels and $*$ means any number of additional dimensions including none. All but the last dimension of the inputs should be the same.
- **input2** - $(*, H_{in2})$ where H_{in2} = in2_channels and $*$ means any number of additional dimensions including none. All but the last dimension of the inputs should be the same.
- **output** - $(*, H_{out})$ where H_{out} = out_channels and $*$ means any number of additional dimensions including none. All but the last dimension are the same shape as the inputs.

Dtype:

- **input1** (Tensor) - The dtype must be float16 or float32 and be same as **input2** .
- **input2** (Tensor) - The dtype must be float16 or float32 and be same as **input1** .
- **output** (Tensor) - With the same dtype as the inputs.

Weights:

- **weight** (Parameter) - The learnable weights with shape (out_channels, in1_channels, in2_channels). When *weight_init* is `None` , the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in1\_channels}}$.
- **bias** (Parameter) - The learnable bias of shape (out_channels). If *has_bias* is `True` and *bias_init* is `None` , the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in1\_channels}}$.

Raises

- **`TypeError`** –If *in1_channels*, *in2_channels* or *out_channels* is not an int.
- **`TypeError`** –If *has_bias* is not a bool.
- **`ValueError`** –If length of shape of *weight_init* is not equal to 3 or shape[0] of *weight_init* is not equal to *out_channels* or shape[1] of *weight_init* is not equal to *in1_channels* or shape[2] of *weight_init* is not equal to *in2_channels*.
- **`ValueError`** –If length of shape of *bias_init* is not equal to 1 or shape[0] of *bias_init* is not equal to *out_channels*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x1 = Tensor(np.random.randn(128, 20), mindspore.float32)
>>> x2 = Tensor(np.random.randn(128, 30), mindspore.float32)
>>> net = nn.BiDense(20, 30, 40)
>>> output = net(x1, x2)
>>> print(output.shape)
(128, 40)
```

3.10 Dropout Layer

API Name	Description	Supported Platforms
mindspore.nn.Dropout	Dropout layer for the input.	`Ascend` `GPU` `CPU`
mindspore.nn.Dropout1d	During training, randomly zeroes entire channels of the input tensor with probability p from a Bernoulli distribution (For a 3-dimensional tensor with a shape of (N, C, L), the channel feature map refers to a 1-dimensional feature map with the shape of L).	`Ascend` `GPU` `CPU`

continues on next page

Table 10 – continued from previous page

mindspore.nn.Dropout2d	During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of $NCHW$, the channel feature map refers to a 2-dimensional feature map with the shape of HW).	Ascend GPU CPU
mindspore.nn.Dropout3d	During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 5-dimensional tensor with a shape of $NCDHW$, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).	Ascend GPU CPU

3.10.1 mindspore.nn.Dropout

class mindspore.nn.**Dropout**(*keep_prob=0.5*, *p=None*, *dtype=mstype.float32*)

Dropout layer for the input.

Dropout is a means of regularization that reduces overfitting by preventing correlations between neuronal nodes. The operator randomly sets some neurons output to 0 according to *p*, which means the probability of discarding during training. And the return will be multiplied by $\frac{1}{1-p}$ during training. During the reasoning, this layer returns the same Tensor as the *x*.

This technique is proposed in paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting and proved to be effective to reduce over-fitting and prevents neurons from co-adaptation. See more details in Improving neural networks by preventing co-adaptation of feature detectors.

Note:

- Each channel will be zeroed out independently on every construct call.
- Parameter *keep_prob* will be removed in a future version, please use parameter *p* instead. Parameter *p* means the probability of the element of the input tensor to be zeroed.
- Parameter *dtype* will be removed in a future version. It is not recommended to define this parameter.

Parameters

- **keep_prob** (*float*) –Deprecated. The keep rate, greater than 0 and less equal than 1. E.g. rate=0.9, dropping out 10% of input neurons. Default: `0.5`.
- **p** (*Union[float, int, None]*) –The dropout rate, greater than or equal to 0 and less than 1. E.g. rate=0.9, dropping out 90% of input neurons. Default: `None`.
- **dtype** (*mindspore.dtype*) –Data type of *input*. Default: `mstype.float32`.

Inputs:

- **x** (Tensor) - The input of Dropout with data type of float16 or float32. The shape of *x* cannot be less than 1.

Outputs:

Tensor, output tensor with the same shape as the *x*.

Raises

- **TypeError** –If *keep_prob* is not a float.

- **TypeError** – If the dtype of *p* is not float or int.
- **TypeError** – If dtype of *x* is not neither float16 nor float32.
- **ValueError** – If *keep_prob* is not in range (0, 1].
- **ValueError** – If *p* is not in range [0, 1).
- **ValueError** – If length of shape of *x* is less than 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.ones([2, 2, 3]), mindspore.float32)
>>> net = nn.Dropout(p=0.2)
>>> net.set_train()
>>> output = net(x)
>>> print(output.shape)
(2, 2, 3)
```

3.10.2 mindspore.nn.Dropout1d

class mindspore.nn.**Dropout1d**(*p=0.5*)

During training, randomly zeroes entire channels of the input tensor with probability *p* from a Bernoulli distribution (For a 3-dimensional tensor with a shape of (N, C, L), the channel feature map refers to a 1-dimensional feature map with the shape of L).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *1D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call with probability *p* using samples from a Bernoulli distribution.

The paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting mentioned this technology, And it is proved that it can effectively reduce over fitting and prevent neuronal coadaptation. For more details, refer to Improving neural networks by preventing co-adaptation of feature detectors .

Dropout1d can improve the independence between channel feature maps.

Parameters

p (*float, optional*) – The dropping probability of a channel, between 0 and 1, e.g. $p = 0.8$, which means an 80% chance of being set to 0. Default: `0.5` .

Inputs:

- **x** (Tensor) - A tensor with shape (N, C, L) or (C, L), where N is the batch size, C is the number of channels, L is the feature length. The data type must be int8, int16, int32, int64, float16, float32 or float64.

Outputs:

Tensor, has the same shape and data type as *x*.

Raises

- **TypeError** – If *x* is not a Tensor.

- **TypeError** – If the data type of *p* is not float.
- **ValueError** – If *p* is out of the range *[0.0, 1.0]*.
- **ValueError** – If the shape of *x* is not *2D* or *3D*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> op = ms.nn.Dropout1d(p=0.6)
>>> op.training = True
>>> a = ms.Tensor(np.ones((3, 3)), ms.float32)
>>> output = op(a)
```

3.10.3 mindspore.nn.Dropout2d

class mindspore.nn.**Dropout2d**(*p=0.5*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of *NCHW*, the channel feature map refers to a 2-dimensional feature map with the shape of *HW*).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *2D* tensor input[i,j]. At each forward propagation, each channel will be independently determined to be set to zero with probability *p*.

Dropout2d can improve the independence between channel feature maps.

Refer to *mindspore.ops.dropout2d()* for more details.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> dropout = nn.Dropout2d(p=0.5)
>>> x = Tensor(np.ones([2, 1, 2, 3]), mindspore.float32)
>>> output = dropout(x)
>>> print(output.shape)
(2, 1, 2, 3)
```

3.10.4 mindspore.nn.Dropout3d

class mindspore.nn.**Dropout3d**(*p=0.5*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 5-dimensional tensor with a shape of $NCDHW$, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).

For example, the j_th channel of the i_th sample in the batched input is a to-be-processed *3D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call which based on Bernoulli distribution probability *p*.

Dropout3d can improve the independence between channel feature maps.

Refer to *mindspore.ops.dropout3d()* for more details.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> dropout = nn.Dropout3d(p=0.5)
>>> x = Tensor(np.ones([2, 1, 2, 1, 2]), mindspore.float32)
>>> output = dropout(x)
>>> print(output.shape)
(2, 1, 2, 1, 2)
```

3.11 Normalization Layer

API Name	Description	Supported Platforms
mindspore.nn.BatchNorm1d	This layer applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D or 2D inputs) to reduce internal covariate shift.	`Ascend` `GPU` `CPU`
mindspore.nn.BatchNorm2d	Batch Normalization is widely used in convolutional networks.	`Ascend` `GPU` `CPU`
mindspore.nn.BatchNorm3d	Batch Normalization is widely used in convolutional networks.	`Ascend` `GPU` `CPU`
mindspore.nn.GroupNorm	Group Normalization over a mini-batch of inputs.	`Ascend` `GPU` `CPU`
mindspore.nn.InstanceNorm1d	This layer applies Instance Normalization over a 3D input (a mini-batch of 1D inputs with additional channel dimension).	`GPU`
mindspore.nn.InstanceNorm2d	This layer applies Instance Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension).	`GPU`
mindspore.nn.InstanceNorm3d	This layer applies Instance Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension).	`GPU`
mindspore.nn.LayerNorm	Applies Layer Normalization over a mini-batch of inputs.	`Ascend` `GPU` `CPU`

continues on next page

Table 11 – continued from previous page

mindspore.nn.SyncBatchNorm	Sync Batch Normalization layer over a N-dimension input. Ascend

3.11.1 mindspore.nn.BatchNorm1d

class mindspore.nn.**BatchNorm1d**(*num_features*, *eps=1e-5*, *momentum=0.9*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *moving_mean_init='zeros'*, *moving_var_init='ones'*, *use_batch_statistics=None*, *data_format='NCHW'*, *dtype=mstype.float32*)

This layer applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D or 2D inputs) to reduce internal covariate shift. Batch Normalization is widely used in convolutional networks. For the setailed contents, refer to Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Note: The implementation of BatchNorm is different in graph mode and pynative mode, therefore the mode is not recommended to be changed after net was initialized.

Parameters

- **num_features** (*int*) –number of features or channels C of the input x .
- **eps** (*float*) –ϵ added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float*) –A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.9` .
- **affine** (*bool*) –A bool value. When set to `True` , γ and β can be learned. Default: `True` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the γ weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the β weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'`, etc. Default: `'zeros'` .
- **moving_mean_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the moving mean. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'zeros'` .
- **moving_var_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the moving variance. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .
- **use_batch_statistics** (*bool*) –If `true` , use the mean value and variance value of current batch data. If `false` , use the mean value and variance value of specified value. If `None` , the training process will use the mean and variance of current batch data and track the running mean and variance, the evaluation process will use the running mean and variance. Default: `None` .
- **data_format** (*str*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C) or (N, C, L) , where N is the batch size, C is the number of features or channels, and L is the sequence length. Supported types: float16, float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape (N, C) or (N, C, L) .

Raises

- **TypeError** – If *num_features* is not an int.
- **TypeError** – If *eps* is not a float.
- **ValueError** – If *num_features* is less than 1.
- **ValueError** – If *momentum* is not in range [0, 1].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> net = ms.nn.BatchNorm1d(num_features=4)
>>> x = ms.Tensor(np.array([[0.7, 0.5, 0.5, 0.6],
...                         [0.5, 0.4, 0.6, 0.9]]).astype(np.float32))
>>> output = net(x)
>>> print(output)
[[ 0.6999965   0.4999975  0.4999975  0.59999704 ]
 [ 0.4999975   0.399998   0.59999704 0.89999545 ]]
```

3.11.2 mindspore.nn.BatchNorm2d

class mindspore.nn.**BatchNorm2d**(*num_features*, *eps=1e-5*, *momentum=0.9*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *moving_mean_init='zeros'*, *moving_var_init='ones'*, *use_batch_statistics=None*, *data_format='NCHW'*, *dtype=mstype.float32*)

Batch Normalization is widely used in convolutional networks. This layer applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) to avoid internal covariate shift as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Note: The implementation of BatchNorm is different in graph mode and pynative mode, therefore that mode can not be changed after net was initialized. Note that the formula for updating the *moving_mean* and *moving_var* is

$$\text{moving\_mean} = \text{moving\_mean} * \text{momentum} + * (1 - \text{momentum})$$

$$\text{moving\_var} = \text{moving\_var} * \text{momentum} + {}^{2} * (1 - \text{momentum})$$

where *moving_mean* is the updated mean, *moving_var* is the updated variance, , ^{2} are the observed value (mean and variance respectively) of each batch of data.

Parameters

- **num_features** (*int*) –The number of channels of the input tensor. Expected input size is (N, C, H, W), C represents the number of channels.
- **eps** (*float, optional*) –ϵ added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float, optional*) –A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.9` .
- **affine** (*bool, optional*) –A bool value. When set to `True` , γ and β can be learned. Default: `True` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the γ weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the β weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'zeros'` .
- **moving_mean_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the moving mean. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'zeros'` .
- **moving_var_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the moving variance. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .
- **use_batch_statistics** (*bool, optional*) –Default: `None` .
  - If `true` , use the mean value and variance value of current batch data and track running mean and running variance.
  - If `false` , use the mean value and variance value of specified value, and not track statistical value.
  - If `None` , the use_batch_statistics is automatically set to `true` or `false` according to the training and evaluation mode. During training, the parameter is set to true, and during evaluation, the parameter is set to false.
- **data_format** (*str, optional*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C, H, W). Supported types: float16, float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape (N, C, H, W).

Raises

- **TypeError** –If *num_features* is not an int.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].
- **ValueError** –If *data_format* is neither `'NHWC'` not `'NCHW'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> net = ms.nn.BatchNorm2d(num_features=3)
>>> x = ms.Tensor(np.ones([1, 3, 2, 2]).astype(np.float32))
>>> output = net(x)
>>> print(output)
[[[[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]
  [[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]
  [[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]]]
```

3.11.3 mindspore.nn.BatchNorm3d

class mindspore.nn.**BatchNorm3d**(*num_features*, *eps=1e-5*, *momentum=0.9*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *moving_mean_init='zeros'*, *moving_var_init='ones'*, *use_batch_statistics=None*, *dtype=mstype.float32*)

Batch Normalization is widely used in convolutional networks. This layer applies Batch Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) to avoid internal covariate shift.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Note: The implementation of BatchNorm is different in graph mode and pynative mode, therefore that mode can not be changed after net was initialized. Note that the formula for updating the running_mean and running_var is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times x_t + \text{momentum} \times \hat{x}$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Parameters

- **num_features** (*int*) – C from an expected input of size (N, C, D, H, W) .
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float*) – A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.9` .
- **affine** (*bool*) – A bool value. When set to `True` , gamma and beta can be learned. Default: `True` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the gamma weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the beta weight. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'zeros'` .
- **moving_mean_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the moving mean. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'zeros'` .
- **moving_var_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the moving variance. The values of str refer to the function mindspore.common.initializer including `'zeros'` , `'ones'` , etc. Default: `'ones'` .

- **use_batch_statistics** (*bool*) –If true, use the mean value and variance value of current batch data. If `false`, use the mean value and variance value of specified value. If `None` , the training process will use the mean and variance of current batch data and track the running mean and variance, the evaluation process will use the running mean and variance. Default: `None` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$. Supported types: float16, float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape $(N, C_{out}, D_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** –If *num_features* is not an int.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> net = ms.nn.BatchNorm3d(num_features=3)
>>> x = ms.Tensor(np.ones([16, 3, 10, 32, 32]).astype(np.float32))
>>> output = net(x)
>>> print(output.shape)
(16, 3, 10, 32, 32)
```

3.11.4 mindspore.nn.GroupNorm

class mindspore.nn.**GroupNorm**(*num_groups*, *num_channels*, *eps=1e-05*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *dtype=mstype.float32*)

Group Normalization over a mini-batch of inputs.

Group Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Group Normalization. Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization. γ and β are scale and shift values obtained by training learning. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Parameters

- **num_groups** (*int*) –The number of groups to be divided along the channel dimension.
- **num_channels** (*int*) –The number of input channels.

- **eps** (*float*) –A value added to the denominator for numerical stability. Default: `1e-05` .
- **affine** (*bool*) –A bool value, this layer will have learnable affine parameters when set to `true` . Default: `True` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the gamma weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , `'xavier_uniform'` , `'he_uniform'` , etc. Default: `'ones'` . If gamma_init is a Tensor, the shape must be (*num_channels*).
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the beta weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , `'xavier_uniform'` , `'he_uniform'` , etc. Default: `'zeros'` . If beta_init is a Tensor, the shape must be (*num_channels*).
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - The input feature with shape $(N, C, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the *x*.

Raises

- **TypeError** –If *num_groups* or *num_channels* is not an int.
- **TypeError** –If *eps* is not a float.
- **TypeError** –If *affine* is not a bool.
- **ValueError** –If *num_groups* or *num_channels* is less than 1.
- **ValueError** –If *num_channels* is not divided by *num_groups*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> group_norm_op = ms.nn.GroupNorm(2, 2)
>>> x = ms.Tensor(np.ones([1, 2, 4, 4], np.float32))
>>> output = group_norm_op(x)
>>> print(output)
[[[[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
  [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]]]
```

3.11.5 mindspore.nn.InstanceNorm1d

class mindspore.nn.**InstanceNorm1d**(*num_features*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *dtype=mstype.float32*)

This layer applies Instance Normalization over a 3D input (a mini-batch of 1D inputs with additional channel dimension). Refer to the paper Instance Normalization: The Missing Ingredient for Fast Stylization. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The size of γ and β, learnable parameters vectors, is num_features if affine is True. The standard-deviation is calculated via the biased estimator.

This layer uses instance statistics computed from input data in both training and evaluation modes.

InstanceNorm1d and BatchNorm1d are very similar, but have some differences. InstanceNorm1d is applied on each channel of channeled data like RGB images, but BatchNorm1d is usually applied on each batch of batched data.

Note: Note that the formula for updating the running_mean and running_var is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times x_t + \text{momentum} \times \hat{x}$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Parameters

- **num_features** (*int*) – C from an expected input of size (N, C, L).
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: `1e-5`.
- **momentum** (*float*) – A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.1`.
- **affine** (*bool*) – A bool value. When set to True, gamma and beta can be learned. Default: `True`.
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the gamma weight. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, etc. When initialized with Tensor, the shape should be (C). Default: `'ones'`.
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the beta weight. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, etc. When initialized with Tensor, the shape should be (C). Default: `'zeros'`.
- **dtype** (*mindspore.dtype*) – Dtype of Parameters. Default: `mstype.float32`.

Inputs:

- **x** (Tensor) - Tensor of shape (N, C, L). Data type: float16 or float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape (N, C, L). Same type and shape as the x.

Raises

- **TypeError** – If the type of *num_features* is not int.
- **TypeError** – If the type of *eps* is not float.
- **TypeError** – If the type of *momentum* is not float.
- **TypeError** – If the type of *affine* is not bool.

- **TypeError** –If the type of *gamma_init*/*beta_init* is not same, or if the initialized element type is not float32.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].
- **ValueError** –If the shape of *gamma_init* / *beta_init* is not (C).
- **KeyError** –If any of *gamma_init*/*beta_init* is str and there is no homonymous class inheriting from *Initializer*.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.InstanceNorm1d(3)
>>> x = ms.Tensor(np.ones([2, 3, 5]), ms.float32)
>>> output = net(x)
>>> print(output.shape)
(2, 3, 5)
```

3.11.6 mindspore.nn.InstanceNorm2d

class mindspore.nn.**InstanceNorm2d**(*num_features*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *dtype=mstype.float32*)

This layer applies Instance Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension). Refer to the paper Instance Normalization: The Missing Ingredient for Fast Stylization. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

γ and β are learnable parameter vectors of size num_features if affine is True. The standard-deviation is calculated via the biased estimator.

This layer uses instance statistics computed from input data in both training and evaluation modes.

InstanceNorm2d and BatchNorm2d are very similar, but have some differences. InstanceNorm2d is applied on each channel of channeled data like RGB images, but BatchNorm2d is usually applied on each batch of batched data.

Note: Note that the formula for updating the running_mean and running_var is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times x_t + \text{momentum} \times \hat{x}$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Parameters

- **num_features** (*int*) –C from an expected input of size (N, C, H, W).
- **eps** (*float*) –A value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float*) –A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.1` .
- **affine** (*bool*) –A bool value. When set to `True` , gamma and beta can be learned. Default: `True` .

- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the gamma weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , etc. When initialized with Tensor, the shape should be (C). Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the beta weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , etc. When initialized with Tensor, the shape should be (C). Default: `'zeros'` .
- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C, H, W). Data type: float16 or float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape (N, C, H, W). Same type and shape as the *x*.

Raises

- **TypeError** –If the type of *num_features* is not int.
- **TypeError** –If the type of *eps* is not float.
- **TypeError** –If the type of *momentum* is not float.
- **TypeError** –If the type of *affine* is not bool.
- **TypeError** –If the type of *gamma_init*/*beta_init* is not same, or if the initialized element type is not float32.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].
- **ValueError** –If the shape of *gamma_init* / *beta_init* is not (C).
- **KeyError** –If any of *gamma_init*/*beta_init* is str and there is no homonymous class inheriting from *Initializer*.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.InstanceNorm2d(3)
>>> x = ms.Tensor(np.ones([2, 3, 2, 2]), ms.float32)
>>> output = net(x)
>>> print(output.shape)
(2, 3, 2, 2)
```

3.11.7 mindspore.nn.InstanceNorm3d

class mindspore.nn.**InstanceNorm3d**(*num_features*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *dtype=mstype.float32*)

This layer applies Instance Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension). Refer to the paper Instance Normalization: The Missing Ingredient for Fast Stylization. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

γ and β are learnable parameter vectors of size num_features if affine is True. The standard-deviation is calculated via the biased estimator.

This layer uses instance statistics computed from input data in both training and evaluation modes.

InstanceNorm3d and BatchNorm3d are very similar, but have some differences. InstanceNorm3d is applied on each channel of channeled data like RGB images, but BatchNorm3d is usually applied on each batch of batched data.

Note: Note that the formula for updating the running_mean and running_var is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times x_t + \text{momentum} \times \hat{x}$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Parameters

- **num_features** (*int*) – C from an expected input of size (N, C, D, H, W).
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float*) – A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.1` .
- **affine** (*bool*) – A bool value. When set to `True` , gamma and beta can be learned. Default: `True` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the gamma weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , etc. When initialized with Tensor, the shape should be (C). Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initializer for the beta weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , etc. When initialized with Tensor, the shape should be (C). Default: `'zeros'` .
- **dtype** (*mindspore.dtype*) – Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C, D, H, W). Data type: float16 or float32.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape (N, C, D, H, W). Same type and shape as the x.

Raises

- **TypeError** – If the type of *num_features* is not int.
- **TypeError** – If the type of *eps* is not float.
- **TypeError** – If the type of *momentum* is not float.
- **TypeError** – If the type of *affine* is not bool.

- **TypeError** –If the type of *gamma_init*/*beta_init* is not same, or if the initialized element type is not float32.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].
- **ValueError** –If the shape of *gamma_init* / *beta_init* is not (C).
- **KeyError** –If any of *gamma_init*/*beta_init* is str and the homonymous class inheriting from *Initializer* not exists.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> net = ms.nn.InstanceNorm3d(3)
>>> x = ms.Tensor(np.ones([2, 3, 5, 2, 2]), ms.float32)
>>> output = net(x)
>>> print(output.shape)
(2, 3, 5, 2, 2)
```

3.11.8 mindspore.nn.LayerNorm

class mindspore.nn.**LayerNorm**(*normalized_shape*, *begin_norm_axis=- 1*, *begin_params_axis=- 1*, *gamma_init='ones'*, *beta_init='zeros'*, *epsilon=1e-7*, *dtype=mstype.float32*)

Applies Layer Normalization over a mini-batch of inputs.

Layer Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Layer Normalization. Unlike Batch Normalization, Layer Normalization performs exactly the same computation at training and testing time. It is applied across all channels and pixel but only one batch size. γ and β are trainable scale and shift. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Parameters

- **normalized_shape** (*Union(tuple[int], list[int])*) –The normalization is performed over axis *begin_norm_axis* $\cdots$ *R - 1*. R is the dimension size of input *x*.
- **begin_norm_axis** (*int*) –The first normalization dimension: normalization will be performed along dimensions *begin_norm_axis: R*, the value should be in [-1, R). Default: `-1` .
- **begin_params_axis** (*int*) –The begin axis of the parameter input (γ, β) to apply LayerNorm, the value should be in [-1, R). Default: `-1` .
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the γ weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , `'xavier_uniform'` , `'he_uniform'` , etc. Default: `'ones'` .
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number]*) –Initializer for the β weight. The values of str refer to the function *initializer* including `'zeros'` , `'ones'` , `'xavier_uniform'` , `'he_uniform'` , etc. Default: `'zeros'` .
- **epsilon** (*float*) –A value added to the denominator for numerical stability(ϵ). Default: `1e-7` .

- **dtype** (*mindspore.dtype*) –Dtype of Parameters. Default: `mstype.float32` .

Inputs:

- **x** (Tensor) - The shape of x is $(x_1, x_2, ..., x_R)$, and *input_shape[begin_norm_axis:]* is equal to *normalized_shape*.

Outputs:

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the x.

Raises

- **TypeError** –If *normalized_shape* is neither a list nor tuple.
- **TypeError** –If *begin_norm_axis* or *begin_params_axis* is not an int.
- **TypeError** –If *epsilon* is not a float.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.ones([20, 5, 10, 10]), ms.float32)
>>> shape1 = x.shape[1:]
>>> m = ms.nn.LayerNorm(shape1,  begin_norm_axis=1, begin_params_axis=1)
>>> output = m(x).shape
>>> print(output)
(20, 5, 10, 10)
```

3.11.9 mindspore.nn.SyncBatchNorm

class mindspore.nn.**SyncBatchNorm**(*num_features*, *eps=1e-5*, *momentum=0.9*, *affine=True*, *gamma_init='ones'*, *beta_init='zeros'*, *moving_mean_init='zeros'*, *moving_var_init='ones'*, *use_batch_statistics=None*, *process_groups=None*, *dtype=mstype.float32*)

Sync Batch Normalization layer over a N-dimension input.

Sync Batch Normalization is cross device synchronized Batch Normalization. The implementation of Batch Normalization only normalizes the data within each device. Sync Batch Normalization will normalize the input within the group. It has been described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Note: Currently, SyncBatchNorm only supports 2D and 4D inputs. γ and β are trainable scale and shift.

Parameters

- **num_features** (*int*) –C from an expected input of size (N, C, H, W).
- **eps** (*float, optional*) –ϵ, a value added to the denominator for numerical stability. Default: `1e-5` .

- **momentum** (*float, optional*) –A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.9`.
- **affine** (*bool, optional*) –A bool value. When set to `True`, γ and β are learnable parameters. When set to `False`, γ and β are unlearnable parameters. Default: `True`.
- **gamma_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the γ weight. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, `'xavier_uniform'`, `'he_uniform'`, etc. Default: `'ones'`.
- **beta_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –Initializer for the β weight. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, `'xavier_uniform'`, `'he_uniform'`, etc. Default: `'zeros'`.
- **moving_mean_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initializer for the moving mean. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, `'xavier_uniform'`, `'he_uniform'`, etc. Default: `'zeros'`.
- **moving_var_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) – Initializer for the moving variance. The values of str refer to the function *initializer* including `'zeros'`, `'ones'`, `'xavier_uniform'`, `'he_uniform'`, etc. Default: `'ones'`.
- **use_batch_statistics** (*bool, optional*) –If `true`, use the mean value and variance value of current batch data. If `false`, use the mean value and variance value of specified value. If `None`, training process will use the mean and variance of current batch data and track the running mean and variance, eval process will use the running mean and variance. Default: `None`.
- **process_groups** (*list, optional*) –A list to divide devices into different sync groups, containing N subtraction lists. Each subtraction list contains int numbers identifying rank ids which need to be synchronized in the same group. All int values must be in [0, rank_size) and different from each other. Default: `None`, indicating synchronization across all devices.
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `mstype.float32`.

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape $(N, C_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** –If *num_features* is not an int.
- **TypeError** –If *eps* is not a float.
- **TypeError** –If *process_groups* is not a list.
- **ValueError** –If *num_features* is less than 1.
- **ValueError** –If *momentum* is not in range [0, 1].
- **ValueError** –If rank_id in *process_groups* is not in range [0, rank_size).

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Please see the Ascend tutorial for more details.

For the GPU devices, users need to prepare the host file and mpi, please see the mpirun Startup .

For the CPU device, users need to write a dynamic cluster startup script, please see the Dynamic Cluster Startup .

This example should be run with multiple devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> ms.reset_auto_parallel_context()
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.DATA_PARALLEL)
>>> sync_bn_op = ms.nn.SyncBatchNorm(num_features=3, process_groups=[[0, 1], [2, 3]])
>>> x = ms.Tensor(np.ones([1, 3, 2, 2]), ms.float32)
>>> output = sync_bn_op(x)
>>> print(output)
[[[[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]
  [[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]
  [[ 0.999995 0.999995 ]
   [ 0.999995 0.999995 ]]]]
```

3.12 Pooling Layer

API Name	Description	Supported Platforms
mindspore.nn.AdaptiveAvgPool1d	Applies a 1D adaptive average pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.nn.AdaptiveAvgPool2d	This operator applies a 2D adaptive average pooling to an input signal composed of multiple input planes.	Ascend GPU CPU
mindspore.nn.AdaptiveAvgPool3d	This operator applies a 3D adaptive average pooling to an input signal composed of multiple input planes.	Ascend GPU CPU
mindspore.nn.AdaptiveMaxPool1d	Applies a 1D adaptive maximum pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.nn.AdaptiveMaxPool2d	This operator applies a 2D adaptive max pooling to an input signal composed of multiple input planes.	Ascend GPU CPU
mindspore.nn.AdaptiveMaxPool3d	Calculates the 3D adaptive max pooling for an input Tensor.	GPU CPU

continues on next page

Table 12 – continued from previous page

mindspore.nn.AvgPool1d	Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.	Ascend GPU CPU
mindspore.nn.AvgPool2d	Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.	Ascend GPU CPU
mindspore.nn.AvgPool3d	Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes.	Ascend GPU CPU
mindspore.nn.FractionalMaxPool3d	Applies the 3D FractionalMaxPool operation over *input*.	GPU CPU
mindspore.nn.LPPool1d	Applying 1D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.	Ascend GPU CPU
mindspore.nn.LPPool2d	Applying 2D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.	Ascend GPU CPU
mindspore.nn.MaxPool1d	Applies a 1D max pooling over an input Tensor which can be regarded as a composition of 1D planes.	Ascend GPU CPU
mindspore.nn.MaxPool2d	Applies a 2D max pooling over an input Tensor which can be regarded as a composition of 2D planes.	Ascend GPU CPU
mindspore.nn.MaxPool3d	3D max pooling operation.	Ascend GPU CPU
mindspore.nn.MaxUnpool1d	Computes the inverse of mindspore.nn.MaxPool1d.	GPU CPU
mindspore.nn.MaxUnpool2d	Computes the inverse of mindspore.nn.MaxPool2d.	GPU CPU
mindspore.nn.MaxUnpool3d	Computes the inverse of mindspore.nn.MaxPool3d.	GPU CPU

3.12.1 mindspore.nn.AdaptiveAvgPool1d

class mindspore.nn.**AdaptiveAvgPool1d**(*output_size*)

Applies a 1D adaptive average pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically, the input is of shape (N_{in}, C_{in}, L_{in}), AdaptiveAvgPool1d outputs regional average in the L_{in}-dimension. The output is of shape $(N_{in}, C_{in}, L_{out})$, where L_{out} is defined by *output_size*.

Note: L_{in} must be divisible by *output_size*.

Parameters

output_size (*int*) –the target output size L_{out}.

Inputs:

- **input** (Tensor) - Tensor of shape (N, C_{in}, L_{in}), with float16 or float32 data type.

Outputs:

Tensor of shape (N, C_{in}, L_{out}), has the same type as *input*.

Raises

- **TypeError** –If *output_size* is not an int.
- **TypeError** –If *input* is neither float16 nor float32.
- **ValueError** –If *output_size* is less than 1.
- **ValueError** –If length of shape of *input* is not equal to 3.
- **ValueError** –If the last dimension of *input* is smaller than *output_size*.
- **ValueError** –If the last dimension of *input* is not divisible by *output_size*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.AdaptiveAvgPool1d(output_size=2)
>>> input = ms.Tensor(np.random.randint(0, 10, [1, 3, 6]), ms.float32)
>>> output = pool(input)
>>> result = output.shape
>>> print(result)
(1, 3, 2)
```

3.12.2 mindspore.nn.AdaptiveAvgPool2d

class mindspore.nn.**AdaptiveAvgPool2d**(*output_size*)

This operator applies a 2D adaptive average pooling to an input signal composed of multiple input planes. That is, for any input size, the size of the specified output is H x W. The number of output features is equal to the number of input features.

The input and output data format can be "NCHW" and "CHW". N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

$$
\begin{aligned}
h_{start} &= floor(i * H_{in}/H_{out}) \\
h_{end} &= ceil((i+1) * H_{in}/H_{out}) \\
w_{start} &= floor(j * W_{in}/W_{out}) \\
w_{end} &= ceil((j+1) * W_{in}/W_{out}) \\
Output(i,j) &= \frac{\sum Input[h_{start}:h_{end}, w_{start}:w_{end}]}{(h_{end}-h_{start})*(w_{end}-w_{start})}
\end{aligned}
$$

Parameters

output_size (*Union[int, tuple]*) –The target output size is H x W. *output_size* can be a tuple consisted of int type H and W, or a single H for H x H, or None. If it is None, it means the output size is the same as the input size.

Inputs:

- **input** (Tensor) - The input of AdaptiveAvgPool2d, which is a 3D or 4D tensor, with float16, float32 or float64 data type.

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$.

Raises

- **ValueError** –If *output_size* is a tuple and the length of *output_size* is not 2.
- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **ValueError** –If the dimension of *input* is less than or equal to the dimension of *output_size*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.AdaptiveAvgPool2d(2)
>>> input_x = ms.Tensor(np.array([[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                               [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                               [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]), ms.
↪float32)
>>> output = pool(input_x)
>>> result = output.shape
>>> print(result)
(3, 2, 2)
```

3.12.3 mindspore.nn.AdaptiveAvgPool3d

class mindspore.nn.**AdaptiveAvgPool3d**(*output_size*)

This operator applies a 3D adaptive average pooling to an input signal composed of multiple input planes. That is, for any input size, the size of the specified output is (D, H, W). The number of output features is equal to the number of input planes.

Suppose the last 3 dimension size of input is (inD, inH, inW), then the last 3 dimension size of output is $(outD, outH, outW)$.

$$
\begin{aligned}
&\forall \quad od \in [0, outD-1], oh \in [0, outH-1], ow \in [0, outW-1] \\
&output[od, oh, ow] = \\
&\qquad mean(input[istartD : iendD+1, istartH : iendH+1, istartW : iendW+1]) \\
&where, \\
&\qquad istartD = \left\lceil \frac{od * inD}{outD} \right\rceil \\
&\qquad iendD = \left\lfloor \frac{(od+1) * inD}{outD} \right\rfloor \\
&\qquad istartH = \left\lceil \frac{oh * inH}{outH} \right\rceil \\
&\qquad iendH = \left\lfloor \frac{(oh+1) * inH}{outH} \right\rfloor \\
&\qquad istartW = \left\lceil \frac{ow * inW}{outW} \right\rceil \\
&\qquad iendW = \left\lfloor \frac{(ow+1) * inW}{outW} \right\rfloor
\end{aligned}
$$

Parameters

output_size (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (D, H, W), or an int D for (D, D, D). D, H and W can be int or None which means the output size is the same as that of the input.

Inputs:

- **input** (Tensor) - The input of AdaptiveAvgPool3d, which is a 5D or 4D Tensor, with float16, float32 or float64 data type.

Outputs:

Tensor, with the same type as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **ValueError** –If the dimension of *input* is not 4D or 5D.
- **ValueError** –If *output_size* value is not positive.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> # case 1: output_size=(3, 3, 4)
>>> output_size=(3, 3, 4)
>>> input_x_val = np.random.randn(4, 3, 5, 6, 7)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = ms.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(4, 3, 3, 3, 4)
>>> # case 2: output_size=4
>>> output_size=5
>>> input_x_val = np.random.randn(2, 3, 8, 6, 12)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = ms.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(2, 3, 5, 5, 5)
>>> # case 3: output_size=(None, 4, 5)
>>> output_size=(None, 4, 5)
>>> input_x_val = np.random.randn(4, 1, 9, 10, 8)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = ms.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(4, 1, 9, 4, 5)
```

3.12.4 mindspore.nn.AdaptiveMaxPool1d

class mindspore.nn.**AdaptiveMaxPool1d**(*output_size*)

Applies a 1D adaptive maximum pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically, the input is of shape (N_{in}, C_{in}, L_{in}), AdaptiveMaxPool1d outputs regional maximum in the L_{in}-dimension. The output is of shape $(N_{in}, C_{in}, L_{out})$, where L_{out} is defined by *output_size*.

Note: L_{in} must be divisible by *output_size*.

Parameters

output_size (*int*) –the target output size L_{out}.

Inputs:

- **x** (Tensor) - Tensor of shape (N, C_{in}, L_{in}), with float16 or float32 data type.

Outputs:

Tensor of shape (N, C_{in}, L_{out}), has the same type as x.

Raises

- **TypeError** –If x is neither float16 nor float32.
- **TypeError** –If *output_size* is not an int.
- **ValueError** –If *output_size* is less than 1.
- **ValueError** –If the last dimension of x is smaller than *output_size*.
- **ValueError** –If the last dimension of x is not divisible by *output_size*.
- **ValueError** –If length of shape of x is not equal to 3.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.AdaptiveMaxPool1d(output_size=3)
>>> x = ms.Tensor(np.random.randint(0, 10, [1, 3, 6]), ms.float32)
>>> output = pool(x)
>>> result = output.shape
>>> print(result)
(1, 3, 3)
```

3.12.5 mindspore.nn.AdaptiveMaxPool2d

class mindspore.nn.**AdaptiveMaxPool2d**(*output_size*, *return_indices=False*)

This operator applies a 2D adaptive max pooling to an input signal composed of multiple input planes. That is, for any input size, the size of the specified output is H x W. The number of output features is equal to the number of input planes.

The input and output data format can be "NCHW" and "CHW". N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

For max adaptive pool2d:

$$
\begin{aligned}
h_{start} &= floor(i * H_{in}/H_{out}) \\
h_{end} &= ceil((i+1) * H_{in}/H_{out}) \\
w_{start} &= floor(j * W_{in}/W_{out}) \\
w_{end} &= ceil((j+1) * W_{in}/W_{out}) \\
Output(i,j) &= \max Input[h_{start}:h_{end}, w_{start}:w_{end}]
\end{aligned}
$$

Note: In KBK mode, *output_size* does not support mutable.

Parameters

- **output_size** (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (H, W), or an int H for (H, H). H and W can be int or None. If it is None, it means the output size is the same as the input size.
- **return_indices** (*bool, optional*) –Whether to output the index of the maximum value. If *return_indices* is `True` , the indices of max value would be output. Default: `False` .

Inputs:

- **input** (Tensor) - The input of AdaptiveMaxPool2d, which is a 3D or 4D tensor, with float16, float32 or float64 data type.

Outputs:

Tensor, with the same type as the *input*. Shape of the output is $input\_shape[: len(input\_shape) - len(out\_shape)] + out\_shape$.

Raises

- **TypeError** –If *output_size* is not int or tuple.
- **TypeError** –If *input* is not a tensor.
- **TypeError** –If *return_indices* is not a bool.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **ValueError** –If *output_size* is a tuple and the length of *output_size* is not 2.
- **ValueError** –If the dimension of *input* is not NCHW or CHW.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> # case 1: output_size=(None, 2)
>>> input = ms.Tensor(np.array([[[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                              [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                              [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]]), ms.
↪float32)
>>> adaptive_max_pool_2d = ms.nn.AdaptiveMaxPool2d((None, 2))
>>> output = adaptive_max_pool_2d(input)
>>> print(output)
[[[[2. 3.]
   [5. 6.]
   [8. 9.]]
  [[2. 3.]
   [5. 6.]
   [8. 9.]]
  [[2. 3.]
```

(continues on next page)

(continued from previous page)

```
   [5. 6.]
   [8. 9.]]]]
>>> # case 2: output_size=2
>>> adaptive_max_pool_2d = ms.nn.AdaptiveMaxPool2d(2)
>>> output = adaptive_max_pool_2d(input)
>>> print(output)
[[[[5. 6.]
   [8. 9.]]
  [[5. 6.]
   [8. 9.]]
  [[5. 6.]
   [8. 9.]]]]
>>> # case 3: output_size=(1, 2)
>>> adaptive_max_pool_2d = ms.nn.AdaptiveMaxPool2d((1, 2))
>>> output = adaptive_max_pool_2d(input)
>>> print(output)
[[[[8. 9.]]
  [[8. 9.]]
  [[8. 9.]]]]
```

3.12.6 mindspore.nn.AdaptiveMaxPool3d

class mindspore.nn.**AdaptiveMaxPool3d**(*output_size*, *return_indices=False*)

Calculates the 3D adaptive max pooling for an input Tensor. That is, for any input size, the size of the specified output is (D, H, W).

Parameters

- **output_size** (*Union[int, tuple]*) –The specified output size, which is a positive integer that represents depth, height and width, or a tuple of three positive integers that represent depth, height and width respectively. If it is None, the output size and input size of the corresponding dimension are the same.
- **return_indices** (*bool, optional*) –If *return_indices* is `True` , the indices of max value would be output. Otherwise, the indices will not be returned. Default: `False` .

Inputs:

- **input** (Tensor) - Tensor, has shape of (C, D, H, W) or (N, C, D, H, W).

Outputs:

- **y** (Tensor) - Tensor, has the same number of dims and data type as the *input* .
- **argmax** (Tensor) - Tensor, the indices of the maximum values along with the outputs, has the same shape as *y* and a dtype of int32. Return this only when *return_indices* is `True` .

Raises

- **TypeError** –If *input* is not a Tensor.
- **ValueError** –If the dimensions number of *input* is not 4 or 5.
- **TypeError** –If dtype of *input* is not int, uint or float.
- **ValueError** –If *output_size* is neither an int nor a tuple with shape $(3,)$.

Supported Platforms:

`GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> input = ms.Tensor(np.arange(0,36).reshape((1, 3, 3, 4)).astype(np.float32))
>>> output_size = (1, 1, 2)
>>> net = ms.nn.AdaptiveMaxPool3d(output_size, True)
>>> output = net(input)
>>> print(output[0].asnumpy())
[[[[33. 35.]]]]
>>> print(output[1].asnumpy())
[[[[33 35]]]]
```

3.12.7 mindspore.nn.AvgPool1d

class mindspore.nn.**AvgPool1d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically the input is of shape (N_{in}, C_{in}, L_{in}), AvgPool1d outputs regional average in the (L_{in})-dimension. Given *kernel_size* l_{ker} and *stride* s_0, the operation is as follows:

$$\text{output}(N_i, C_j, l) = \frac{1}{l_{ker}} \sum_{n=0}^{l_{ker}-1} \text{input}(N_i, C_j, s_0 \times l + n)$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **kernel_size** (*int, optional*) – The size of kernel window used to take the average value, Default: `1` .
- **stride** (*int, optional*) – The distance of kernel moving, an int number that represents the width of movement is strides, Default: `1` .
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input at the begin and end so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it is uniformly distributed around the input, if it is odd, the excess padding is goes to the right side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible length. If a full stride cannot be formed, the extra pixels will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding at the begin and end is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int]), optional*) – Pooling padding value, only `"pad"` mode can be set to non-zero. Default: `0` . padding can only be an integer or a tuple/list containing a single integer, in which case padding times or padding[0] times are padded on both sides of the input.
- **ceil_mode** (*bool, optional*) – If `True` , use ceil to compute the output shape instead of floor. Default: `False` .

- **count_include_pad** (*bool, optional*) – If `True` , averaging calculation will include the zero-padding. Default: `True` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C_{in}, L_{in}) or (C_{in}, L_{in}).

Outputs:

Tensor of shape (N, C_{out}, L_{out}) or (C_{out}, L_{out}).

If *pad_mode* is in *pad* mode, the output shape calculation formula is as follows:

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

Raises

- **TypeError** – If *kernel_size* or *stride* is not an int.
- **ValueError** – If *pad_mode* is not `"valid"` , `"same"` or `"pad"` with not case sensitive.
- **ValueError** – If *kernel_size* or *strides* is less than 1.
- **ValueError** – If length of *padding* tuple/list is not 1.
- **ValueError** – If length of shape of *x* is not equal to 2 or 3.
- **ValueError** – If *padding* is non-zero when *pad_mode* is not `"pad"`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.AvgPool1d(kernel_size=6, stride=1)
>>> x = ms.Tensor(np.random.randint(0, 10, [1, 3, 6]), ms.float32)
>>> output = pool(x)
>>> result = output.shape
>>> print(result)
(1, 3, 1)
>>> pool2 = ms.nn.AvgPool1d(4, stride=1, ceil_mode=True, pad_mode="pad", padding=2)
>>> x1 = ms.ops.randn(6, 6, 8)
>>> output = pool2(x1)
>>> print(output.shape)
(6, 6, 9)
```

3.12.8 mindspore.nn.AvgPool2d

class mindspore.nn.**AvgPool2d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*, *data_format='NCHW'*)

Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.

Typically the input is of shape $(N_{in}, C_{in}, H_{in}, W_{in})$, AvgPool2d outputs regional average in the (H_{in}, W_{in})-dimension. Given kernel size $ks = (h_{ker}, w_{ker})$ and stride $s = (s_0, s_1)$, the operation is as follows:

$$\text{output}(N_i, C_j, h, w) = \frac{1}{h_{ker} * w_{ker}} \sum_{m=0}^{h_{ker}-1} \sum_{n=0}^{w_{ker}-1} \text{input}(N_i, C_j, s_0 \times h + m, s_1 \times w + n)$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the average value. The data type of kernel_size must be int or a single element tuple and the value represents the height and width, or a tuple of two int numbers that represent height and width respectively. Default: `1` .
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number or a single element tuple that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int])*) –Pooling padding value, only `"pad"` mode can be set to non-zero. Default: `0` . *padding* can only be an integer or a tuple/list containing one or two integers. If *padding* is an integer or a tuple/list containing one integer, it will be padded *padding* times in the four directions of the input. If *padding* is a tuple/list containing two integers, it will be padded *padding[0]* times in the up-down direction of the input and *padding[1]* times in the left-right direction of the input.
- **ceil_mode** (*bool*) –If `True` , use ceil to compute the output shape instead of floor. Default: `False` .
- **count_include_pad** (*bool*) –If `True` , averaging calculation will include the zero-padding. Default: `True` .
- **divisor_override** (*int*) –If it is specified as a non-zero parameter, this parameter will be used as the divisor in the average calculation. Otherwise, *kernel_size* will be used as the divisor. Default: `None` .
- **data_format** (*str*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}).

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$.

If *pad_mode* is in *pad* mode, the output shape calculation formula is as follows:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

Raises

- **TypeError** – If *kernel_size* or *strides* is neither int nor tuple.
- **ValueError** – If *pad_mode* is not `"valid"` , `"same"` or `"pad"` with not case sensitive.
- **ValueError** – If *data_format* is neither `'NCHW'` nor `'NHWC'`.
- **ValueError** – If *padding*, *ceil_mode*, *count_include_pad*, or *divisor_override* is used, or *pad_mode* is `"pad"` when *data_format* is 'NHWC'.
- **ValueError** – If *kernel_size* or *strides* is less than 1.
- **ValueError** – If length of *padding* tuple/list is not 1 or 2.
- **ValueError** – If length of shape of *x* is not equal to 3 or 4.
- **ValueError** – If *divisor_override* is less than or equal to 0.
- **ValueError** – If *padding* is non-zero when *pad_mode* is not `"pad"`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.AvgPool2d(kernel_size=3, stride=1)
>>> x = ms.Tensor(np.random.randint(0, 10, [1, 2, 4, 4]), ms.float32)
>>> output = pool(x)
>>> print(output.shape)
(1, 2, 2, 2)
>>> x = ms.ops.randn(6, 6, 8, 8)
>>> pool2 = ms.nn.AvgPool2d(4, stride=1, pad_mode="pad", padding=2, divisor_override=5)
>>> output2 = pool2(x)
>>> print(output2.shape)
(6, 6, 9, 9)
```

3.12.9 mindspore.nn.AvgPool3d

class mindspore.nn.**AvgPool3d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*)

Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes. Typically, the input is of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$, and AvgPool3D outputs regional average in the (D_{in}, H_{in}, W_{in})-dimension. Given kernel size is $ks = (d_{ker}, h_{ker}, w_{ker})$ and stride $s = (s_0, s_1, s_2)$, the operation is as follows.

Warning: *kernel_size* is in the range [1, 255]. *stride* is in the range [1, 63].

$$\text{output}(N_i, C_j, d, h, w) = \frac{1}{d_{ker} * h_{ker} * w_{ker}} \sum_{l=0}^{d_{ker}-1} \sum_{m=0}^{h_{ker}-1} \sum_{n=0}^{w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **kernel_size** (*Union[int, tuple[int]], optional*) – The size of kernel used to take the average value, can be an int number or a single element tuple that represents depth, height and width, or a tuple of three positive integers that represent depth, height and width respectively. Default: `1` .
- **stride** (*Union[int, tuple[int]], optional*) – The distance of kernel moving, can be a positive int or a single element tuple that represents the depth, height and width of movement, or a tuple of three positive integers that represents depth, height and width of movement respectively. If the value is None, the default value *kernel_size* is used. Default: `1` .
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int]), optional*) – Pooling padding value, only `"pad"` mode can be set to non-zero. Default: `0` . Only the following paddings are supported:
  - *padding* is an integer or a tuple/list containing one integer, it will be padded in six directions of front, back, top, bottom, left and right of the input.
  - *padding* is a tuple/list containing three integers, it will be padded in front and back of the input *padding[0]* times, up and down *padding[1]* times, and left and right of the input *padding[2]* times.
- **ceil_mode** (*bool, optional*) – If `True` , use ceil to compute the output shape instead of floor. Default: `False` .
- **count_include_pad** (*bool, optional*) – If `True` , averaging calculation will include the zero-padding. Default: `True` .

- **divisor_override** (*int, optional*) – If it is specified as a non-zero parameter, this parameter will be used as the divisor in the average calculation. Otherwise, *kernel_size* will be used as the divisor. Default: `None` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$. Currently support float16, float32 and float64 data type.

Outputs:

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$, with the same data type as x.

If *pad_mode* is in *pad* mode, the output shape calculation formula is as follows:

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{kernel\_size}[2]}{\text{stride}[2]} + 1 \right\rfloor$$

Raises

- **TypeError** – If *kernel_size* is neither an int nor a tuple.
- **TypeError** – If *stride* is neither an int nor a tuple.
- **TypeError** – If *padding* is neither an int nor a tuple/list.
- **TypeError** – If *ceil_mode* or *count_include_pad* is not a bool.
- **TypeError** – If *divisor_override* is not an int.
- **ValueError** – If numbers in *kernel_size* or *stride* are not positive.
- **ValueError** – If *kernel_size* or *stride* is a tuple whose length is not equal to 3.
- **ValueError** – If *padding* is a tuple/list whose length is neither 1 nor 3.
- **ValueError** – If element of *padding* is less than 0.
- **ValueError** – If length of shape of *x* is neither 4 nor 5.
- **ValueError** – If *divisor_override* is less than or equal to 0.
- **ValueError** – If *padding* is non-zero when *pad_mode* is not `"pad"`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> pool = ms.nn.AvgPool3d(kernel_size=3, stride=1)
>>> x = ms.ops.randn(1, 2, 4, 4, 5).astype(ms.float32)
>>> output = pool(x)
>>> print(output.shape)
(1, 2, 2, 2, 3)
>>> x1 = ms.ops.randn(6, 5, 7, 7, 5).astype(ms.float32)
```

(continues on next page)

(continued from previous page)

```
>>> pool2 = ms.nn.AvgPool3d(4, stride=2, pad_mode="pad", padding=(2, 2, 1), divisor_
↪override=10)
>>> output2 = pool2(x1)
>>> print(output2.shape)
(6, 5, 4, 4, 2)
```

3.12.10 mindspore.nn.FractionalMaxPool3d

class mindspore.nn.**FractionalMaxPool3d**(*kernel_size*, *output_size=None*, *output_ratio=None*, *return_indices=False*, *_random_samples=None*)

Applies the 3D FractionalMaxPool operation over *input*. The output Tensor shape can be determined by either *output_size* or *output_ratio*, and the step size is determined by *_random_samples*. *output_size* will take effect when *output_size* and *output_ratio* are set at the same time. And *output_size* and *output_ratio* can not be `None` at the same time.

Refer to the paper Fractional MaxPooling by Ben Graham for more details.

The input and output data format can be "NCDHW". N is the batch size, C is the number of channels, D the feature depth, H is the feature height, and W is the feature width.

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, is a positive int that represents depth, height and width of the kernel, or a tuple of three positive integers that represent depth, height and width respectively.
- **output_size** (*Union[int, tuple[int]], optional*) – The shape of the target *output_size*, is an int number that represents depth, height and width, or a tuple of three positive integers that represents depth, height and width respectively. If `None` , the shape of the target will be determined by *output_ratio*. Default: `None` .
- **output_ratio** (*Union[float, tuple[float]], optional*) – The ratio of target output shape to input shape. Specifying the size of the output tensor by using a ratio of the input size. Data type : float16, float32, float64, and value is between (0, 1). If `None` , the shape of the target will be determined by *output_size*.Default: `None` .
- **return_indices** (*bool, optional*) – Whether to return the indices of max value. Default: `False` .
- **_random_samples** (*Tensor, optional*) – The random step of FractionalMaxPool3d, which is a 3D tensor. Tensor of data type: float16, float32, double, and value is between [0, 1). Supported shape $(N, C, 3)$ or $(1, C, 3)$. Default: `None`, the values of *_random_samples* will be randomly distributed using uniform distribution over an interval [0,1).

Inputs:

- **input** (Tensor) - The input of FractionalMaxPool3d, which is a 4D or 5D tensor. Tensor of data type : float16, float32, float64. Supported shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$.

Outputs:

- **y** (Tensor) - A tensor, the output of FractionalMaxPool3d. Has the same data type with *input*. Has the shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$, where $(D_{out}, H_{out}, W_{out})$ = *output_size* or $(D_{out}, H_{out}, W_{out})$ = *output_ratio* * (D_{in}, H_{in}, W_{in}) .
- **argmax** (Tensor) - The indices along with the outputs, which is a Tensor, with the same shape as the *y* and int32 data type. It will output only when *return_indices* is True.

Raises

- **TypeError** – If *input* is not a 4D or 5D tensor.

- **TypeError** – If *_random_samples* is not a 3D tensor.
- **TypeError** – If data type of *imput_x* is not float16, float32, float64.
- **TypeError** – If dtype of *_random_samples* is not float16, float32, float64.
- **TypeError** – If dtype of *argmax* is not int32, int64.
- **TypeError** – if _random_samples to have the different dtypes as input.
- **ValueError** – If *output_size* is a tuple and if *output_size* length is not 3.
- **ValueError** – If *kernel_size* is a tuple and if *kernel_size* length is not 3.
- **ValueError** – If numbers in *output_size* or *kernel_size* is not positive.
- **ValueError** – if *output_size* and *output_ratio* are None at the same time.
- **ValueError** – If the first dimension size of *input* and *_random_samples* is not equal.
- **ValueError** – If the second dimension size of *input* and *_random_samples* is not equal.
- **ValueError** – If the third dimension size of *_random_samples* is not 3.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = ms.Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
...               .reshape([1, 1, 2, 2, 4]), ms.float32)
>>> _random_samples = ms.Tensor(np.array([0.7, 0.7, 0.7]).reshape([1, 1, 3]), ms.float32)
>>> net = ms.nn.FractionalMaxPool3d(kernel_size=(1, 1, 1), output_size=(1, 1, 3),
...                                 _random_samples=_random_samples, return_indices=True)
>>> output, argmax = net(x)
>>> print(output)
[[[[[13. 14. 16.]]]]]
>>> print(argmax)
[[[[[12 13 15]]]]]
>>> net = ms.nn.FractionalMaxPool3d(kernel_size=(1, 1, 1), output_ratio=(0.5, 0.5, 0.5),
...                                 _random_samples=_random_samples, return_indices=True)
>>> output, argmax = net(x)
>>> print(output)
[[[[[13. 16.]]]]]
>>> print(argmax)
[[[[[12 15]]]]]
```

3.12.11 mindspore.nn.LPPool1d

class mindspore.nn.**LPPool1d**(*norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applying 1D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.

Typically the input is of shape (N_{in}, C_{in}, L_{in}) or (C_{in}, L_{in}), the output is of shape $(N_{out}, C_{out}, L_{out})$ or (C_{out}, L_{out}), with the same shape as input, the operation is as follows.

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **norm_type** (*Union[int, float]*) –Type of normalization, represents p in the formula, can not be 0.
  - if p = 1, the result is the sum of the elements within the pooling kernel(proportional to average pooling).
  - if p = ∞, the result is the result of maximum pooling.
- **kernel_size** (*int*) –The size of kernel window.
- **stride** (*int*) –The distance of kernel moving, an int number that represents the width of movement is stride, if the value is None, the default value *kernel_size* is used. Default: `None` .
- **ceil_mode** (*bool*) –If `True`, use ceil to calculate output shape. If `False`, use ceil to calculate output shape. Default: `False` .

Inputs:

- **x** (Tensor) - Tensor of shape (N_{in}, C_{in}, L_{in}) or (C_{in}, L_{in}).

Outputs:

- **output** (Tensor) - LPPool1d result, with shape $(N_{out}, C_{out}, L_{out})$ or (C_{out}, L_{out}), it has the same data type as x, where

$$L_{out} = \left\lfloor \frac{L_{in} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

Raises

- **TypeError** –If x is not a Tensor.
- **TypeError** –If *kernel_size* or *stride* is not an int.
- **TypeError** –If *ceil_mode* is not a bool.
- **TypeError** –If *norm_type* is neither float nor int.
- **ValueError** –If *norm_type* is equal to 0.
- **ValueError** –If *kernel_size* or *stride* is less than 1.
- **ValueError** –If length of shape of x is not equal to 2 or 3.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> a = ms.Tensor(np.arange(2 * 3 * 4).reshape((2, 3, 4)), dtype=ms.float32)
>>> net = ms.nn.LPPool1d(norm_type=1, kernel_size=3, stride=1)
>>> out = net(a)
>>> print(out)
[[[ 3.  6.]
  [15. 18.]
  [27. 30.]]
 [[39. 42.]
  [51. 54.]
  [63. 66.]]]
```

3.12.12 mindspore.nn.LPPool2d

class mindspore.nn.**LPPool2d**(*norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applying 2D LPPooling operation on an input Tensor can be regarded as forming a 1D input plane.

Typically the input is of shape (N, C, H_{in}, W_{in}), the output is of shape (N, C, H_{in}, W_{in}), with the same shape as input, the operation is as follows.

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **norm_type** (*Union[int, float]*) –Type of normalization, represents p in the formula, can not be 0.
  - if p = 1, the result is the sum of the elements within the pooling kernel(proportional to average pooling).
  - if p = ∞, the result is the result of maximum pooling.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel window. The data type of kernel_size must be int and the value represents the height and width, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively, if the value is `None`, the default value *kernel_size* is used. Default: `None` .
- **ceil_mode** (*bool*) –Whether to use ceil or floor to calculate output shape. Default: `False` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C, H_{in}, W_{in}).

Outputs:

- **output** (Tensor) - LPPool2d result, with shape (N, C, H_{in}, W_{in}), It has the same data type as x, where

$$H_{out} = \left\lfloor \frac{H_{in} - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *kernel_size* or *stride* is neither int nor tuple.
- **TypeError** – If *ceil_mode* is not a bool.
- **TypeError** – If *norm_type* is neither float nor int.
- **ValueError** – If *norm_type* is equal to 0.
- **ValueError** – If *kernel_size* or *stride* is less than 1.
- **ValueError** – If *kernel_size* or *stride* is a tuple whose length is not equal to *2*.
- **ValueError** – If length of shape of *x* is not equal to 4.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> a = ms.Tensor(np.arange(2 * 3 * 4 * 5).reshape((2, 3, 4, 5)), dtype=ms.float32)
>>> net = ms.nn.LPPool2d(norm_type=1, kernel_size=3, stride=1)
>>> out = net(a)
>>> print(out)
[[[[  54.   63.   72.]
   [  99.  108.  117.]]
  [[ 234.  243.  252.]
   [ 279.  288.  297.]]
  [[ 414.  423.  432.]
   [ 459.  468.  477.]]]
 [[[ 594.  603.  612.]
   [ 639.  648.  657.]]
  [[ 774.  783.  792.]
   [ 819.  828.  837.]]
  [[ 954.  963.  972.]
   [ 999. 1008. 1017.]]]]
```

3.12.13 mindspore.nn.MaxPool1d

class mindspore.nn.**MaxPool1d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

Applies a 1D max pooling over an input Tensor which can be regarded as a composition of 1D planes.

Typically the input is of shape (N_{in}, C_{in}, L_{in}), MaxPool1d outputs regional maximum in the (L_{in})-dimension. Given *kernel size* $ks = (l_{ker})$ and *stride* $s = (s_0)$, the operation is as follows:

$$\text{output}(N_i, C_j, l) = \max_{n=0,\ldots,l_{ker}-1} \text{input}(N_i, C_j, s_0 \times l + n)$$

Parameters

- **kernel_size** (*int, optional*) – The size of kernel used to take the max value. Default: `1` .
- **stride** (*int, optional*) – The distance of kernel moving, an int number that represents the width of movement is stride. Default: `1` .
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input at the begin and end so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it is uniformly distributed around the input, if it is odd, the excess padding is goes to the right side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible length. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding at the begin and end is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int]), optional*) – Padding value for the pooling. Default value is `0`. padding can only be an integer or a tuple/list containing a single integer, in which case padding times or padding[0] times are padded on both sides of the input.
- **dilation** (*Union(int, tuple[int]), optional*) – The spacing between the elements of the kernel in convolution, used to increase the receptive field of the pooling operation. If it is a tuple, its length can only be 1. Default: `1` .
- **return_indices** (*bool, optional*) – If `True` , the function will return both the result of max pooling and the indices of the max elements. Default: `False` .
- **ceil_mode** (*bool, optional*) – If True, use ceil to compute the output shape instead of floor. Default: `False` .

Inputs:

- **x** (Tensor) - Tensor of shape (N, C_{in}, L_{in}) or (C_{in}, L_{in}).

Outputs:

If *return_indices* is `False`, output is a Tensor, with shape (N, C_{out}, L_{out}) or (C_{out}, L_{out}). It has the same data type as *x*.

If *return_indices* is `True`, output is a Tuple of 2 Tensors, representing the maxpool result and where the max values are generated.

- **output** (Tensor) - Maxpooling result, with shape (N, C_{out}, L_{out}) or (C_{out}, L_{out}). It has the same data type as *x*.
- **argmax** (Tensor) - Index corresponding to the maximum value. Data type is int64.

If *pad_mode* is in *pad* mode, the output shape calculation formula is as follows:

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

Raises

- **TypeError** – If *kernel_size* or *strides* is not an int.
- **ValueError** – If *pad_mode* is not `"valid"`, `"same"` or `"pad"`, case-insensitive.
- **ValueError** – If *data_format* is neither `'NCHW'` nor `'NHWC'`.
- **ValueError** – If *kernel_size* or *strides* is less than 1.
- **ValueError** – If length of shape of *x* is not equal to 2 or 3.

- **ValueError** –If *pad_mode* is not "pad", *padding*, *dilation*, *return_indices*, *ceil_mode* parameters are not set to their default values.
- **ValueError** –If the length of the tuple/list *padding* parameter is not 1.
- **ValueError** –If The length of the tuple dilation parameter is not 1.
- **ValueError** –If dilation parameter is neither an integer nor a tuple.
- **ValueError** –If *padding* is non-zero when *pad_mode* is not "pad".

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> mpool1 = nn.MaxPool1d(kernel_size=3, stride=1)
>>> x = ms.Tensor(np.random.randint(0, 10, [1, 2, 4]), ms.float32)
>>> output = mpool1(x)
>>> result = output.shape
>>> print(result)
(1, 2, 2)
>>> np_x = np.random.randint(0, 10, [5, 3, 4])
>>> x = ms.Tensor(np_x, ms.float32)
>>> mpool2 = nn.MaxPool1d(kernel_size=2, stride=1, pad_mode="pad", padding=1, dilation=1,␣
↪return_indices=True)
>>> output = mpool2(x)
>>> print(output[0].shape)
(5, 3, 5)
>>> print(output[1].shape)
(5, 3, 5)
```

3.12.14 mindspore.nn.MaxPool2d

class mindspore.nn.**MaxPool2d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*, *data_format='NCHW'*)

Applies a 2D max pooling over an input Tensor which can be regarded as a composition of 2D planes.

Typically the input is of shape $(N_{in}, C_{in}, H_{in}, W_{in})$, MaxPool2d outputs regional maximum in the (H_{in}, W_{in})-dimension. Given kernel size (h_{ker}, w_{ker}) and stride (s_0, s_1), the operation is as follows.

$$\text{output}(N_i, C_j, h, w) = \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times h + m, s_1 \times w + n)$$

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the max value, is an int number or a single element tuple that represents height and width are both kernel_size, or a tuple of two int numbers that represent height and width respectively. Default: 1 .
- **stride** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number or a single element tuple that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: 1 .

- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int])*) –Specifies the padding value of the pooling operation. Default: `0` . *padding* can only be an integer or a tuple/list containing one or two integers. If *padding* is an integer or a tuple/list containing one integer, it will be padded *padding* times in the four directions of the input. If *padding* is a tuple/list containing two integers, it will be padded *padding[0]* times in the up-down direction of the input and *padding[1]* times in the left-right direction of the input.
- **dilation** (*Union(int, tuple[int])*) –The spacing between the elements of the kernel in convolution, used to increase the receptive field of the pooling operation. If it is a tuple, it must contain one or two integers. Default: `1` .
- **return_indices** (*bool*) –If `True` , the function will return both the result of max pooling and the indices of the max elements. Default: `False` .
- **ceil_mode** (*bool*) –If `True` , use ceil to compute the output shape instead of floor. Default: `False` .
- **data_format** (*str*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}).

Outputs:

If *return_indices* is False, output is a Tensor, with shape (N, C, H_{out}, W_{out}) or $(C_{out}, H_{out}, W_{out})$. It has the same data type as *x*.

If *return_indices* is True, output is a Tuple of 2 Tensors, representing the maxpool result and where the max values are generated.

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$. It has the same data type as *x*.
- **argmax** (Tensor) - Index corresponding to the maximum value. Data type is int64.

If *pad_mode* is in *pad* mode, the output shape calculation formula is as follows:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Raises

- **TypeError** –If *kernel_size* or *stride* is neither int nor tuple.
- **ValueError** –If *pad_mode* is neither `"valid"` nor `"same"` with not case sensitive.
- **ValueError** –If *data_format* is neither `'NCHW'` nor `'NHWC'` .
- **ValueError** –If *kernel_size* or *stride* is less than 1.

- **ValueError** – If length of shape of *x* is not equal to 3 or 4.
- **ValueError** – If *pad_mode* is not "pad", *padding*, *dilation*, *return_indices*, *ceil_mode* parameters are not set to their default values.
- **ValueError** – If the length of the tuple/list *padding* parameter is not 2.
- **ValueError** – If The length of the tuple dilation parameter is not 2.
- **ValueError** – If dilation parameter is neither an integer nor a tuple.
- **ValueError** – If *pad_mode* is "pad" and *data_format* is 'NHWC'.
- **ValueError** – If *padding* is non-zero when *pad_mode* is not "pad".

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pool = ms.nn.MaxPool2d(kernel_size=3, stride=1)
>>> x = ms.Tensor(np.random.randint(0, 10, [1, 2, 4, 4]), ms.float32)
>>> output = pool(x)
>>> print(output.shape)
(1, 2, 2, 2)
>>> np_x = np.random.randint(0, 10, [5, 3, 4, 5])
>>> x = ms.Tensor(np_x, ms.float32)
>>> pool2 = ms.nn.MaxPool2d(kernel_size=2, stride=1, pad_mode="pad", padding=1, dilation=1,␣
↪return_indices=True)
>>> output = pool2(x)
>>> print(output[0].shape)
(5, 3, 5, 6)
>>> print(output[1].shape)
(5, 3, 5, 6)
```

3.12.15 mindspore.nn.MaxPool3d

class mindspore.nn.**MaxPool3d**(*kernel_size=1*, *stride=1*, *pad_mode='valid'*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

3D max pooling operation.

Applies a 3D max pooling over an input Tensor which can be regarded as a composition of 3D planes.

Typically the input is of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$, MaxPool outputs regional maximum in the (D_{in}, H_{in}, W_{in})-dimension. Given kernel size is $ks = (d_{ker}, h_{ker}, w_{ker})$ and stride is $s = (s_0, s_1, s_2)$, the operation is as follows.

$$\text{output}(N_i, C_j, d, h, w) = \max_{l=0,\ldots,d_{ker}-1} \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Note: For Atlas training series products, this interface is not supported.

Parameters

- **kernel_size** (*Union[int, tuple[int]], optional*) –The size of kernel used to take the maximum value, is an int number or a single element tuple that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively. The value must be a positive integer. Default: `1` .
- **stride** (*Union[int, tuple[int]], optional*) –The moving stride of pooling operation, an int number or a single element tuple that represents the moving stride of pooling kernel in the directions of depth, height and the width, or a tuple of three int numbers that represent depth, height and width of movement respectively. The value must be a positive integer. If the value is None, the default value *kernel_size* is used. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *padding* parameter. If this mode is set, *padding* must be greater than or equal to 0.
- **padding** (*Union(int, tuple[int], list[int]), optional*) –Pooling padding value. Default: `0` . *padding* can only be an integer or a tuple/list containing one or three integers. If *padding* is an integer or a tuple/list containing one integer, it will be padded in six directions of front, back, top, bottom, left and right of the input. If *padding* is a tuple/list containing three integers, it will be padded in front and back of the input *padding[0]* times, up and down *padding[1]* times, and left and right of the input *padding[2]* times.
- **dilation** (*Union(int, tuple[int]), optional*) –The spacing between the elements of the kernel in convolution, used to increase the receptive field of the pooling operation. If it is a tuple, it must contain one or three integers. Default: `1` .
- **return_indices** (*bool, optional*) –If `True` , output is a Tuple of 2 Tensors, representing the maxpool result and where the max values are generated. Otherwise, only the maxpool result is returned. Default: `False` .
- **ceil_mode** (*bool, optional*) –If `True`, use ceil to calculate output shape. If `False`, use ceil to calculate output shape. Default: `False` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$ or $(C_{in}, D_{in}, H_{in}, W_{in})$.

Outputs:

If *return_indices* is False, output is a Tensor, with shape $(N_{out}, C_{out}, D_{out}, H_{out}, W_{out})$ or $(C_{out}, D_{out}, H_{out}, W_{out})$. It has the same data type as *x*.

If *return_indices* is True, output is a Tuple of 2 Tensors, representing the maxpool result and where the max values are generated.

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, D_{out}, H_{out}, W_{out})$ or $(C_{out}, D_{out}, H_{out}, W_{out})$. It has the same data type as *x*.
- **argmax** (Tensor) - Index corresponding to the maximum value. Data type is int64.

If *pad_mode* is in `"pad"` mode, the output shape calculation formula is as follows:

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

Raises

- **ValueError** – If length of shape of *x* is not equal to 4 or 5.
- **TypeError** – If *kernel_size* , *stride* , *padding* or *dilation* is neither an int nor a tuple.
- **ValueError** – If *kernel_size* or *stride* is less than 1.
- **ValueError** – If the *padding* parameter is neither an integer nor a tuple of length 3.
- **ValueError** – If *pad_mode* is not set to `"pad"`, setting return_indices to True or dilation to a value other than 1.
- **ValueError** – If *padding* is non-zero when *pad_mode* is not `"pad"`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> import numpy as np
>>> np_x = np.random.randint(0, 10, [5, 3, 4, 6, 7])
>>> x = Tensor(np_x, ms.float32)
>>> pool1 = nn.MaxPool3d(kernel_size=2, stride=1, pad_mode="pad", padding=1, dilation=3,␣
↪return_indices=True)
>>> output = pool1(x)
>>> print(output[0].shape)
(5, 3, 3, 5, 6)
>>> print(output[1].shape)
(5, 3, 3, 5, 6)
>>> pool2 = nn.MaxPool3d(kernel_size=2, stride=1, pad_mode="pad", padding=1, dilation=3,␣
↪return_indices=False)
>>> output2 = pool2(x)
>>> print(output2.shape)
(5, 3, 3, 5, 6)
```

3.12.16 mindspore.nn.MaxUnpool1d

class mindspore.nn.**MaxUnpool1d**(*kernel_size*, *stride=None*, *padding=0*)

Computes the inverse of *mindspore.nn.MaxPool1d*.

MaxUnpool1d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}) or (C, H_{in}), and the output is of shape (N, C, H_{out}) or (C, H_{out}). The operation is as follows.

$$H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value.
- **stride** (*Union[int, tuple[int]]*) – The distance of kernel moving, If stride is None, then stride equal to kernel_size. Default: `None` .

- **padding** (*Union[int, tuple[int]]*) –The pad value to be filled. Default: `0` .

Inputs:

- **x** (Tensor) - The input Tensor to invert. Tensor of shape (N, C, H_{in}) or (C, H_{in}).
- **indices** (Tensor) - Max values' index represented by the indices. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, H_{in} - 1]$. Data type must be in int32 or int64.
- **output_size** (tuple[int], optional) - The output size. Default: `None` . If output_size is `None`, then the shape of output computed by kernel_size, stride and padding. If output_size is not `None`, then output_size must be (N, C, H) , (C, H) or (H) and output_size must belong to $[(N, C, H_{out} - stride[0]), (N, C, H_{out} + stride[0])]$.

Outputs:

Tensor, with shape (N, C, H_{out}) or (C, H_{out}), with the same data type with *x*.

Raises

- **TypeError** –If data type of *x* or *indices* is not supported.
- **TypeError** –If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** –If numbers in *stride*, *padding* (also support 0 and (0)) or *kernel_size* is not positive.
- **ValueError** –If the shapes of *x* and *indices* are not equal.
- **ValueError** –If *x* whose length is not 2 or 3.
- **ValueError** –If type of *output_size* is not tuple.
- **ValueError** –If *output_size* whose length is not 0, 2 or 3.
- **ValueError** –If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.array([[2, 4, 6, 8]]).astype(np.float32))
>>> indices = ms.Tensor(np.array([[1, 3, 5, 7]]).astype(np.int64))
>>> maxunpool1d = ms.nn.MaxUnpool1d(kernel_size =2, stride=2, padding=0)
>>> output = maxunpool1d(x, indices)
>>> print(output.asnumpy())
[[0. 2. 0. 4. 0. 6. 0. 8.]]
```

3.12.17 mindspore.nn.MaxUnpool2d

class mindspore.nn.**MaxUnpool2d**(*kernel_size*, *stride=None*, *padding=0*)

Computes the inverse of *mindspore.nn.MaxPool2d*.

MaxUnpool2d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}), and the output is of shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}). The operation is as follows.

$$H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$
$$W_{out} = (W_{in} - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1]$$

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]]*) – The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. If stride is `None`, then stride equal to kernel_size. Default: `None` .
- **padding** (*Union[int, tuple[int]]*) – The pad value to be filled. Default: `0` . If *padding* is an integer, the paddings of height and width are the same, equal to padding. If *padding* is a tuple of two integers, the padding of height and width equal to padding[0] and padding[1] correspondingly.

Inputs:

- **x** (Tensor) - The input Tensor to invert. Tensor of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- **indices** (Tensor) - Max values' index represented by the indices. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **output_size** (tuple[int], optional) - The output size. Default: `None` . If output_size is `None`, then the shape of output computed by kernel_size, stride and padding. If output_size is not `None`, then output_size must be (N, C, H, W), (C, H, W) or (H, W) and output_size must belong to $[(N, C, H_{out} - stride[0], W_{out} - stride[1]), (N, C, H_{out} + stride[0], W_{out} + stride[1])]$.

Outputs:

Tensor, with shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), with the same data type with *x*.

Raises

- **TypeError** – If data type of *x* or *indices* is not supported.
- **TypeError** – If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** – If numbers in *stride*, *padding* (also support 0 and (0, 0)) or *kernel_size* is not positive.
- **ValueError** – If the shape of *x* and *indices* are not equal.
- **ValueError** – If *kernel_size*, *stride* or *padding* is a tuple whose length is not equal to 2.
- **ValueError** – If *x* whose length is not 3 or 4.
- **ValueError** – If *output_size* whose type is not tuple.
- **ValueError** – If *output_size* whose length is not 0, 3 or 4.
- **ValueError** – If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.array([[[[0, 1], [8, 9]]]]).astype(np.float32))
>>> indices = ms.Tensor(np.array([[[[0, 1], [2, 3]]]]).astype(np.int64))
>>> maxunpool2d = ms.nn.MaxUnpool2d(kernel_size=1, stride=1, padding=0)
>>> output = maxunpool2d(x, indices)
>>> print(output.asnumpy())
[[[[0. 1.]
   [8. 9.]]]]
```

3.12.18 mindspore.nn.MaxUnpool3d

class mindspore.nn.**MaxUnpool3d**(*kernel_size*, *stride=None*, *padding=0*)

Computes the inverse of *mindspore.nn.MaxPool3d*.

MaxUnpool3d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$, and the output is of shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$. The operation is as follows.

$$
\begin{aligned}
D_{out} &= (D_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0] \\
H_{out} &= (H_{in} - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1] \\
W_{out} &= (W_{in} - 1) \times stride[2] - 2 \times padding[2] + kernel\_size[2]
\end{aligned}
$$

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively.
- **stride** (*Union[int, tuple[int]]*) – The distance of kernel moving, an int number that represents the depth, height and width of movement are both stride, or a tuple of three int numbers that represent depth, height and width of movement respectively. If stride is `None`, then stride equal to kernel_size. Default: `None` .
- **padding** (*Union[int, tuple[int]]*) – The pad value to be filled. Default: `0` . If *padding* is an integer, the paddings of depth, height and width are the same, equal to padding. If *padding* is a tuple of three integers, the padding of depth, height and width equal to padding[0], padding[1] and padding[2] correspondingly.

Inputs:

- **x** (Tensor) - The input Tensor to invert. Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$.
- **indices** (Tensor) - Max values' index represented by the indices. Tensor of shape must be same with input 'x'. Values of indices must belong to $[0, D_{in} \times H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **output_size** (tuple[int], optional) - The output size. Default: `None` . If output_size is `None`, then the shape of output computed by kernel_size, stride and padding. If output_size is not `None`, then output_size must be (N, C, D, H, W) , (C, D, H, W) or (D, H, W) and output_size must belong to $[(N, C, D_{out} - stride[0], H_{out} - stride[1], W_{out} - stride[2]), (N, C, D_{out} + stride[0], H_{out} + stride[1], W_{out} + stride[2])]$.

Outputs:

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$, with the same data type with x.

Raises

- **TypeError** – If data type of *x* or *indices* is not supported.
- **TypeError** – If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** – If numbers in *stride* or *padding* (also support 0 and (0, 0, 0)) or *kernel_size* is not positive.
- **ValueError** – If the shape of *x* and *indices* are not equal.
- **ValueError** – If *kernel_size*, *stride* or *padding* is a tuple whose length is not equal to 3.
- **ValueError** – If *x* whose length is not 4 or 5.
- **ValueError** – If *output_size* whose length is not 0, 4 or 5.
- **ValueError** – If *output_size* whose type is not tuple.
- **ValueError** – If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.array([[[[[0, 1], [8, 9]]]]]).astype(np.float32))
>>> indices= ms.Tensor(np.array([[[[[0, 1], [2, 3]]]]]).astype(np.int64))
>>> maxunpool3d = ms.nn.MaxUnpool3d(kernel_size=1, stride=1, padding=0)
>>> output = maxunpool3d(x, indices)
>>> print(output.asnumpy())
[[[[[0. 1.]
    [8. 9.]]]]]
```

3.13 Padding Layer

API Name	Description	Supported Platforms
mindspore.nn.Pad	Pads the input tensor according to the paddings and mode.	Ascend GPU CPU
mindspore.nn.ConstantPad1d	Using a given constant value to pads the last dimensions of input tensor.	Ascend GPU CPU
mindspore.nn.ConstantPad2d	Using a given constant value to pads the last two dimensions of input tensor.	Ascend GPU CPU
mindspore.nn.ConstantPad3d	Using a given constant value to pads the last three dimensions of input tensor.	Ascend GPU CPU
mindspore.nn.ReflectionPad1d	Using a given padding to do reflection pad on the given tensor.	Ascend GPU CPU
mindspore.nn.ReflectionPad2d	Using a given padding to do reflection pad the given tensor.	Ascend GPU CPU
mindspore.nn.ReflectionPad3d	Pad the given tensor in a reflecting way using the input boundaries as the axis of symmetry.	Ascend GPU CPU
mindspore.nn.ReplicationPad1d	Pad on W dimension of input *x* according to *padding*.	GPU

continues on next page

Table 13 – continued from previous page

mindspore.nn.ReplicationPad2d	Pad on HW dimension of input *x* according to *padding*.	GPU
mindspore.nn.ReplicationPad3d	Pad on DHW dimension of input *x* according to *padding*.	GPU
mindspore.nn.ZeroPad2d	Pads the last two dimensions of input tensor with zero.	Ascend GPU CPU

3.13.1 mindspore.nn.Pad

class mindspore.nn.**Pad**(*paddings*, *mode='CONSTANT'*)

Pads the input tensor according to the paddings and mode.

Parameters

- **paddings** (*tuple*) –The shape of parameter *paddings* is $(N, 2)$. N is the rank of input data. All elements of paddings are int type. For D th dimension of the x, paddings[D, 0] indicates how many sizes to be extended ahead of the D th dimension of the input tensor, and paddings[D, 1] indicates how many sizes to be extended behind of the D th dimension of the input tensor. The padded size of each dimension D of the output is: $paddings[D, 0] + input\_x.dim\_size(D) + paddings[D, 1]$, e.g.:

```
mode = "CONSTANT".
paddings = [[1,1], [2,2]].
x = [[1,2,3], [4,5,6], [7,8,9]].
# The above can be seen: 1st dimension of `x` is 3, 2nd dimension of `x` is 3.
# Substitute into the formula to get:
# 1st dimension of output is paddings[0][0] + 3 + paddings[0][1] = 1 + 3 + 1 = 5.
# 2nd dimension of output is paddings[1][0] + 3 + paddings[1][1] = 2 + 3 + 2 = 7.
# So the shape of output is (5, 7).
```

- **mode** (*str*) –Specifies padding mode. The optional values are "CONSTANT" , "REFLECT" , "SYMMETRIC" . Default: "CONSTANT" .

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

Tensor, the tensor after padding.

- If *mode* is "CONSTANT", it fills the edge with 0, regardless of the values of the *x*. If the *x* is [[1,2,3], [4,5,6], [7,8,9]] and *paddings* is [[1,1], [2,2]], then the Outputs is [[0,0,0,0,0,0,0], [0,0,1,2,3,0,0], [0,0,4,5,6,0,0], [0,0,7,8,9,0,0], [0,0,0,0,0,0,0]].
- If *mode* is "REFLECT", it uses a way of symmetrical copying through the axis of symmetry to fill in. If the *x* is [[1,2,3], [4,5,6], [7,8,9]] and *paddings* is [[1,1], [2,2]], then the Outputs is [[6,5,4,5,6,5,4], [3,2,1,2,3,2,1], [6,5,4,5,6,5,4], [9,8,7,8,9,8,7], [6,5,4,5,6,5,4]].
- If *mode* is "SYMMETRIC", the filling method is similar to the "REFLECT". It is also copied according to the symmetry axis, except that it includes the symmetry axis. If the *x* is [[1,2,3], [4,5,6], [7,8,9]] and *paddings* is [[1,1], [2,2]], then the Outputs is [[2,1,1,2,3,3,2], [2,1,1,2,3,3,2], [5,4,4,5,6,6,5], [8,7,7,8,9,9,8], [8,7,7,8,9,9,8]].

Raises

- **TypeError** –If *paddings* is not a tuple.
- **ValueError** –If length of *paddings* is more than 4 or its shape is not $(N, 2)$.

- **ValueError** – If *mode* is not one of "CONSTANT", "REFLECT", "SYMMETRIC".

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn, ops
>>> import numpy as np
>>> # If `mode` is "CONSTANT"
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.pad = nn.Pad(paddings=((1, 1), (2, 2)), mode="CONSTANT")
...     def construct(self, x):
...         return self.pad(x)
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]), mindspore.float32)
>>> pad = Net()
>>> output = pad(x)
>>> print(output)
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 3. 0. 0.]
 [0. 0. 4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
>>> # Another way to call
>>> pad = ops.Pad(paddings=((1, 1), (2, 2)))
>>> # From the above code, we can see following:
>>> # "paddings=((1, 1), (2, 2))",
>>> # paddings[0][0] = 1, indicates a row of values is filled top of the input data in the␣
↪1st dimension.
>>> # Shown as follows:
>>> # [[0. 0. 0.]
>>> #  [1. 2. 3.]
>>> #  [4. 5. 6.]]
>>> # paddings[0][1] = 1 indicates a row of values is filled below input data in the 1st␣
↪dimension.
>>> # Shown as follows:
>>> # [[0. 0. 0.]
>>> #  [1. 2. 3.]
>>> #  [4. 5. 6.]
>>> #  [0. 0. 0.]]
>>> # paddings[1][0] = 2, indicates 2 rows of values is filled in front of input data in the␣
↪2nd dimension.
>>> # Shown as follows:
>>> # [[0. 0. 0. 0. 0.]
>>> #  [0. 0. 1. 2. 3.]
>>> #  [0. 0. 4. 5. 6.]
>>> #  [0. 0. 0. 0. 0.]]
>>> # paddings[1][1] = 2, indicates 2 rows of values is filled in front of input data in the␣
↪2nd dimension.
>>> # Shown as follows:
>>> # [[0. 0. 0. 0. 0. 0. 0.]
>>> #  [0. 0. 1. 2. 3. 0. 0.]
>>> #  [0. 0. 4. 5. 6. 0. 0.]
>>> #  [0. 0. 0. 0. 0. 0. 0.]]
```

(continues on next page)

(continued from previous page)

```
>>> output = pad(x)
>>> print(output)
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 3. 0. 0.]
 [0. 0. 4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
>>> # if mode is "REFLECT"
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.pad = nn.Pad(paddings=((1, 1), (2, 2)), mode="REFLECT")
...     def construct(self, x):
...         return self.pad(x)
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]), mindspore.float32)
>>> pad = Net()
>>> output = pad(x)
>>> print(output)
[[6. 5. 4. 5. 6. 5. 4.]
 [3. 2. 1. 2. 3. 2. 1.]
 [6. 5. 4. 5. 6. 5. 4.]
 [3. 2. 1. 2. 3. 2. 1.]]
>>> # if mode is "SYMMETRIC"
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.pad = nn.Pad(paddings=((1, 1), (2, 2)), mode="SYMMETRIC")
...     def construct(self, x):
...         return self.pad(x)
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6]]), mindspore.float32)
>>> pad = Net()
>>> output = pad(x)
>>> print(output)
[[2. 1. 1. 2. 3. 3. 2.]
 [2. 1. 1. 2. 3. 3. 2.]
 [5. 4. 4. 5. 6. 6. 5.]
 [5. 4. 4. 5. 6. 6. 5.]]
```

3.13.2 mindspore.nn.ConstantPad1d

class mindspore.nn.**ConstantPad1d**(*padding*, *value*)

Using a given constant value to pads the last dimensions of input tensor.

Parameters

- **padding** (*Union[int, tuple]*) –The padding size to pad the last dimension of input tensor. If is int, uses the same padding in both boundaries of input's last dimension. If a 2-tuple, uses (padding_0, padding_1) to pad. If the input is x, the size of last dimension of output is $padding\_0 + x.shape[-1] + padding\_1$. The remaining dimensions of the output are consistent with those of the input. Only support non-negative value while running in Ascend.
- **value** (*Union[int, float]*) –Padding value.

Inputs:

- **x** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions. It is not supported that the size of dimensions is greater than 5 while running on Ascend.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not a tuple or int.
- **TypeError** – If *value* is not int or float.
- **ValueError** – If the length of *padding* with tuple type is not equal to 2.
- **ValueError** – If the output shape after padding is not positive.
- **ValueError** – If the rank of 'x' is more than 5 while running in Ascend.
- **ValueError** – If *padding* contains negative value while running in Ascend.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> # padding is tuple
>>> padding = (0, 1)
>>> value = 0.5
>>> pad1d = ms.nn.ConstantPad1d(padding, value)
>>> out = pad1d(x)
>>> print(out)
[[[[1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]]
  [[1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]]]]
>>> print(out.shape)
(1, 2, 3, 5)
>>> # padding is int
>>> padding = 1
>>> value = 0.5
>>> pad1d = ms.nn.ConstantPad1d(padding, value)
>>> out = pad1d(x)
>>> print(out)
[[[[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]]
  [[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]]]]
>>> print(out.shape)
(1, 2, 3, 6)
```

3.13.3 mindspore.nn.ConstantPad2d

class mindspore.nn.**ConstantPad2d**(*padding*, *value*)

Using a given constant value to pads the last two dimensions of input tensor.

Parameters

- **padding** (*Union[int, tuple]*) – The padding size to pad the last two dimensions of input tensor. If is int, uses the same padding in boundaries of input's last two dimensions. If is tuple and length of padding is 4 uses (padding_0, padding_1, padding_2, padding_3) to pad. If the input is x, the size of last dimension of output is $padding\_0 + x.shape[-1] + padding\_1$. The size of penultimate dimension of output is $padding\_2 + x.shape[-2] + padding\_3$. The remaining dimensions of the output are consistent with those of the input. Only support non-negative value while running in Ascend.
- **value** (*Union[int, float]*) – Padding value.

Inputs:

- **x** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions. It is not supported that the size of dimensions is greater than 5 while running on Ascend.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not a tuple or int.
- **TypeError** – If *value* is not int or float.
- **ValueError** – If the length of *padding* is more than 4 or not a multiple of 2.
- **ValueError** – If the output shape after padding is not positive.
- **ValueError** – If the rank of 'x' is more than 5 while running in Ascend.
- **ValueError** – If *padding* contains negative value while running in Ascend.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1)
>>> value = 0.5
>>> pad2d = ms.nn.ConstantPad2d(padding, value)
>>> out = pad2d(x)
>>> print(out)
[[[[0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  0.5 0.5 0.5 0.5 0.5]]
  [[0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
```

(continues on next page)

(continued from previous page)

```
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  0.5 0.5 0.5 0.5 0.5]]]]
>>> print(out.shape)
(1, 2, 4, 6)
```

3.13.4 mindspore.nn.ConstantPad3d

class mindspore.nn.**ConstantPad3d**(*padding*, *value*)

Using a given constant value to pads the last three dimensions of input tensor.

Parameters

- **padding** (*Union[int, tuple]*) – The padding size to pad the last three dimensions of input tensor. If is int, uses the same padding in boundaries of input's last three dimensions. If is tuple and length of padding is 6 uses (padding_0, padding_1, padding_2, padding_3, padding_4, padding_5) to pad. If the input is x, the size of last dimension of output is $padding\_0 + x.shape[-1] + padding\_1$. The size of penultimate dimension of output is $padding\_2 + x.shape[-2] + padding\_3$. The size of 3rd to last dimension of output is $padding\_4 + x.shape[-3] + padding\_5$. The remaining dimensions of the output are consistent with those of the input. Only support non-negative value while running in Ascend.
- **value** (*Union[int, float]*) – Padding value.

Inputs:

- **x** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions. It is not supported that the size of dimensions is greater than 5 while running on Ascend.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not a tuple or int.
- **TypeError** – If *value* is not int or float.
- **ValueError** – If the length of *padding* is more than 6 or not a multiple of 2.
- **ValueError** – If the output shape after padding is not positive.
- **ValueError** – If the rank of 'x' is more than 5 while running in Ascend.
- **ValueError** – If *padding* contains negative value while running in Ascend.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1, 1, 0)
>>> value = 0.5
>>> pad3d = ms.nn.ConstantPad3d(padding, value)
>>> out = pad3d(x)
>>> print(out)
[[[[0.5 0.5 0.5 0.5 0.5 0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]]
  [[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]]
  [[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]]]]
>>> print(out.shape)
(1, 3, 4, 6)
```

3.13.5 mindspore.nn.ReflectionPad1d

class mindspore.nn.**ReflectionPad1d**(*padding*)

Using a given padding to do reflection pad on the given tensor. 1d means the dimension of padding is 1-dimension.

Parameters

padding (*union[int, tuple]*) – The padding size to pad the last dimension of input tensor. If padding is an integer: all directions will be padded with the same size. If padding is a tuple: uses (pad_left, pad_right) to pad.

Inputs:

- **x** (Tensor) - 2D or 3D, shape: (C, W_{in}) or (N, C, W_{in}).

Outputs:

Tensor, after padding. Shape: (C, W_{out}) or (N, C, W_{out}), where $W_{out} = W_{in} + pad\_left + pad\_right$.

Raises

- **TypeError** – If 'padding' is not a tuple or int.
- **TypeError** – If there is an element in 'padding' that is not int.
- **ValueError** – If the length of 'padding' is not divisible by 2.
- **ValueError** – If there is an element in 'padding' that is negative.
- **ValueError** – If the there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = ms.Tensor(np.array([[[0, 1, 2, 3], [4, 5, 6, 7]]]).astype(np.float32))
>>> # x has shape (1, 2, 4)
>>> padding = (3, 1)
>>> # The first and the second dimension of x remain the same.
>>> # The third dimension of x: W_out = W_in + pad_left + pad_right = 4 + 3 + 1 = 8
>>> pad1d = ms.nn.ReflectionPad1d(padding)
>>> out = pad1d(x)
>>> # The shape of out is (1, 2, 8)
>>> print(out)
[[[3. 2. 1. 0. 1. 2. 3. 2.]
  [7. 6. 5. 4. 5. 6. 7. 6.]]]
```

3.13.6 mindspore.nn.ReflectionPad2d

class mindspore.nn.**ReflectionPad2d**(*padding*)

Using a given padding to do reflection pad the given tensor. 2d means the dimension of padding is 2-dimension.

Parameters

padding (*union[int, tuple]*) –The padding size to pad the input tensor. If padding is an integer: all directions will be padded with the same size. If padding is a tuple: uses $(pad\_left, pad\_right, pad\_up, pad\_down)$ to pad.

Inputs:

- **x** (Tensor) - 3D or 4D, shape: (C, H_{in}, W_{in}) or (N, C, H_{in}, W_{in}).

Outputs:

Tensor, after padding. Shape: (C, H_{out}, W_{out}) or (N, C, H_{out}, W_{out}), where $H_{out} = H_{in} + pad\_up + pad\_down$, $W_{out} = W_{in} + pad\_left + pad\_right$.

Raises

- **TypeError** –If 'padding' is not a tuple or int.
- **TypeError** –If there is an element in 'padding' that is not int.
- **ValueError** –If the length of 'padding' is not divisible by 2.
- **ValueError** –If there is an element in 'padding' that is negative.
- **ValueError** –If the there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = ms.Tensor(np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]]]).astype(np.float32))
>>> # x has shape (1, 3, 3)
>>> padding = (1, 1, 2, 0)
>>> pad2d = ms.nn.ReflectionPad2d(padding)
>>> # The first dimension of x remains the same.
>>> # The second dimension of x: H_out = H_in + pad_up + pad_down = 3 + 1 + 1 = 5
>>> # The third dimension of x: W_out = W_in + pad_left + pad_right = 3 + 2 + 0 = 5
>>> out = pad2d(x)
>>> # The shape of out is (1, 5, 5)
>>> print(out)
[[[7. 6. 7. 8. 7.]
  [4. 3. 4. 5. 4.]
  [1. 0. 1. 2. 1.]
  [4. 3. 4. 5. 4.]
  [7. 6. 7. 8. 7.]]]
```

3.13.7 mindspore.nn.ReflectionPad3d

class mindspore.nn.**ReflectionPad3d**(*padding*)

Pad the given tensor in a reflecting way using the input boundaries as the axis of symmetry. 3d means the dimension of padding is 3-dimension.

Note: ReflectionPad3d has not supported 5D tensor yet.

Parameters

padding (*union[int, tuple]*) – The padding size to pad the input tensor. If padding is an integer: all directions will be padded with the same size. If padding is a tuple: uses $(pad\_left, pad\_right, pad\_up, pad\_down, pad\_front, pad\_back)$ to pad.

Inputs:

- **x** (Tensor) - 4D Tensor, shape: $(N, D_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, after padding. Shape: $(N, D_{out}, H_{out}, W_{out})$, where $D_{out} = D_{in} + pad\_front + pad\_back$, $H_{out} = H_{in} + pad\_up + pad\_down$ $W_{out} = W_{in} + pad\_left + pad\_right$.

Raises

- **TypeError** – If 'padding' is not a tuple or int.
- **TypeError** – If there is an element in 'padding' that is not int.
- **ValueError** – If the length of 'padding' is not divisible by 2.
- **ValueError** – If there is an element in 'padding' that is negative.
- **ValueError** – If the there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> arr = np.arange(8).astype(np.float32).reshape((1, 2, 2, 2))
>>> x = ms.Tensor(arr)
>>> # x has shape (1, 2, 2, 2)
>>> padding = (1, 1, 1, 0, 0, 1)
>>> pad3d = ms.nn.ReflectionPad3d(padding)
>>> out = pad3d(x)
>>> # The first dimension of x remains the same.
>>> # The second dimension of x: D_out = D_in + pad_front + pad_back = 2 + 0 + 1 = 3
>>> # The third dimension of x: H_out = H_in + pad_up + pad_down = 2 + 1 + 0 = 3
>>> # The last dimension of x: W_out = W_in + pad_left + pad_right = 2 + 1 + 1 = 4
>>> # The shape of out is (1, 3, 3, 4)
>>> print(out)
[[[[3. 2. 3. 2.]
   [1. 0. 1. 0.]
   [3. 2. 3. 2.]]
  [[7. 6. 7. 6.]
   [5. 4. 5. 4.]
   [7. 6. 7. 6.]]
  [[3. 2. 3. 2.]
   [1. 0. 1. 0.]
   [3. 2. 3. 2.]]]]
```

3.13.8 mindspore.nn.ReplicationPad1d

class mindspore.nn.**ReplicationPad1d**(*padding*)

Pad on W dimension of input *x* according to *padding*.

Parameters

padding (*union[int, tuple]*) –The padding size to pad the last dimension of *x* .

- If *padding* is an integer, all directions will be padded with the same size.
- If *padding* is a tuple, uses $(pad_{left}, pad_{right})$ to pad.

Inputs:

- **x** (Tensor) - 2D or 3D, shape: (C, W_{in}) or (N, C, W_{in}).

Outputs:

Tensor, after padding. Shape: (C, W_{out}) or (N, C, W_{out}), where $W_{out} = W_{in} + pad_{left} + pad_{right}$

Raises

- **TypeError** –If *padding* is neither a tuple nor an int.
- **TypeError** –If there is an element in *padding* that is not int.
- **ValueError** –If *padding* is tuple and the length of *padding* is not divisible by 2.
- **ValueError** –If *padding* is tuple and there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

GPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad1d = ms.nn.ReplicationPad1d(2)
>>> input = ms.Tensor(np.arange(0, 8).reshape(1, 2, 4), ms.float32)
>>> print(input)
[[[0. 1. 2. 3.]
  [4. 5. 6. 7.]]]
>>> out = pad1d(input)
>>> print(out)
[[[0. 0. 0. 1. 2. 3. 3. 3.]
  [4. 4. 4. 5. 6. 7. 7. 7.]]]
>>> pad1d = ms.nn.ReplicationPad1d((3, 1))
>>> out = pad1d(input)
>>> print(out)
[[[0. 0. 0. 0. 1. 2. 3. 3.]
  [4. 4. 4. 4. 5. 6. 7. 7.]]]
```

3.13.9 mindspore.nn.ReplicationPad2d

class mindspore.nn.**ReplicationPad2d**(*padding*)

Pad on HW dimension of input *x* according to *padding*.

Parameters

padding (*union[int, tuple]*) – The padding size to pad the last two dimension of x .

- If *padding* is an integer, all directions will be padded with the same size.
- If *padding* is a tuple, uses $(pad_{left}, pad_{right}, pad_{up}, pad_{down})$ to pad.

Inputs:

- **x** (Tensor) - 3D or 4D, shape: (C, H_{in}, W_{in}) or (N, C, H_{in}, W_{in}).

Outputs:

Tensor, after padding. Shape: (C, H_{out}, W_{out}) or (N, C, H_{out}, W_{out}), where $H_{out} = H_{in} + pad_{up} + pad_{down}$, $W_{out} = W_{in} + pad_{left} + pad_{right}$.

Raises

- **TypeError** – If *padding* is neither a tuple nor an int.
- **TypeError** – If there is an element in *padding* that is not int.
- **ValueError** – If *padding* is tuple and the length of *padding* is not divisible by 2.
- **ValueError** – If *padding* is tuple and there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

GPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad2d = ms.nn.ReplicationPad2d(2)
>>> input = ms.Tensor(np.arange(0, 9).reshape(1, 1, 3, 3), ms.float32)
>>> print(input)
[[[[0. 1. 2.]
   [3. 4. 5.]
   [6. 7. 8.]]]]
>>> out = pad2d(input)
>>> print(out)
[[[[0. 0. 0. 1. 2. 2. 2.]
   [0. 0. 0. 1. 2. 2. 2.]
   [0. 0. 0. 1. 2. 2. 2.]
   [3. 3. 3. 4. 5. 5. 5.]
   [6. 6. 6. 7. 8. 8. 8.]
   [6. 6. 6. 7. 8. 8. 8.]
   [6. 6. 6. 7. 8. 8. 8.]]]]
>>> pad2d = ms.nn.ReplicationPad2d((1, 1, 2, 0))
>>> out = pad2d(input)
>>> print(out)
[[[[0. 0. 1. 2. 2.]
   [0. 0. 1. 2. 2.]
   [0. 0. 1. 2. 2.]
   [3. 3. 4. 5. 5.]
   [6. 6. 7. 8. 8.]]]]
```

3.13.10 mindspore.nn.ReplicationPad3d

class mindspore.nn.**ReplicationPad3d**(*padding*)

Pad on DHW dimension of input *x* according to *padding*.

Parameters

padding (*union[int, tuple]*) –The padding size to pad the last three dimension of *x* .

- If *padding* is an integer, all directions will be padded with the same size.
- If *padding* is a tuple, uses $(pad_{left}, pad_{right}, pad_{up}, pad_{down}, pad_{front}, pad_{back})$ to pad.

Inputs:

- **x** (Tensor) - 4D or 5D, shape: $(C, D_{in}, H_{in}, W_{in})$ or $(N, C, D_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, after padding, shape: $(C, D_{out}, H_{out}, W_{out})$ or $(N, C, D_{out}, H_{out}, W_{out})$, where $D_{out} = D_{in} + pad_{front} + pad_{back}$, $H_{out} = H_{in} + pad_{up} + pad_{down}$, $W_{out} = W_{in} + pad_{left} + pad_{right}$.

Raises

- **TypeError** –If *padding* is neither a tuple nor an int.
- **TypeError** –If there is an element in *padding* that is not int.
- **ValueError** –If *padding* is tuple and the length of *padding* is not divisible by 2.
- **ValueError** –If *padding* is tuple and there is a dimension mismatch between the padding and the tensor.

Supported Platforms:

GPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad3d = ms.nn.ReplicationPad3d(1)
>>> input = ms.Tensor(np.arange(0, 9).reshape(1, 1, 1, 3, 3), ms.float32)
>>> out = pad3d(input)
>>> print(out)
[[[[[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]
   [[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]
   [[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]]]]
```

3.13.11 mindspore.nn.ZeroPad2d

class mindspore.nn.**ZeroPad2d**(*padding*)

Pads the last two dimensions of input tensor with zero.

Parameters

padding (*Union[int, tuple]*) – The padding size to pad the last two dimensions of input tensor. If is int, uses the same padding in boundaries of input's last two dimensions. If is tuple and length of padding is 4 uses (padding_0, padding_1, padding_2, padding_3) to pad. If the input is x, the size of last dimension of output is $padding\_0 + x.shape[-1] + padding\_1$. The size of penultimate dimension of output is $padding\_2 + x.shape[-2] + padding\_3$. The remaining dimensions of the output are consistent with those of the input. Only support non-negative value while running in Ascend.

Inputs:

- **x** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions. It is not supported that the size of dimensions is greater than 5 while running in Ascend.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not a tuple or int.
- **ValueError** – If the length of *padding* is more than 4 or not a multiple of 2.
- **ValueError** – If the output shape after padding is not positive.

- **ValueError** – If the rank of 'x' is more than 5 while running in Ascend.
- **ValueError** – If *padding* contains negative value while running in Ascend.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1)
>>> pad = ms.nn.ZeroPad2d(padding)
>>> out = pad(x)
>>> print(out)
[[[[0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 0. 0. 0. 0. 0.]]
  [[0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 0. 0. 0. 0. 0.]]]]
>>> print(out.shape)
(1, 2, 4, 6)
```

3.14 Loss Function

API Name	Description	Supported Platforms
mindspore.nn.BCELoss	BCELoss creates a criterion to measure the binary cross entropy between the true labels and predicted labels.	`Ascend` `GPU` `CPU`
mindspore.nn.BCEWithLogitsLoss	Adds sigmoid activation function to input *input* as logits, and uses the given logits to compute binary cross entropy between the *input* and the *target*.	`Ascend` `GPU` `CPU`
mindspore.nn.CosineEmbeddingLoss	CosineEmbeddingLoss creates a criterion to measure the similarity between two tensors using cosine distance.	`Ascend` `GPU` `CPU`
mindspore.nn.CrossEntropyLoss	The cross entropy loss between input and target.	`Ascend` `GPU` `CPU`
mindspore.nn.CTCLoss	Calculates the CTC (Connectionist Temporal Classification) loss.	`Ascend` `GPU` `CPU`
mindspore.nn.DiceLoss	The Dice coefficient is a set similarity loss, which is used to calculate the similarity between two samples.	`Ascend` `GPU` `CPU`
mindspore.nn.FocalLoss	It is a loss function to solve the imbalance of categories and the difference of classification difficulty.	`Ascend`
mindspore.nn.GaussianNLLLoss	Gaussian negative log likelihood loss.	`Ascend` `GPU` `CPU`

continues on next page

Table 14 – continued from previous page

mindspore.nn.HingeEmbeddingLoss	Calculate the Hinge Embedding Loss value based on the input 'logits' and' labels' (only including 1 or -1).	Ascend GPU CPU
mindspore.nn.HuberLoss	HuberLoss calculate the error between the predicted value and the target value.	Ascend GPU CPU
mindspore.nn.KLDivLoss	Computes the Kullback-Leibler divergence between the *logits* and the *labels*.	Ascend GPU CPU
mindspore.nn.L1Loss	L1Loss is used to calculate the mean absolute error between the predicted value and the target value.	Ascend GPU CPU
mindspore.nn.MarginRankingLoss	MarginRankingLoss creates a criterion that measures the loss.	Ascend GPU CPU
mindspore.nn.MAELoss	MAELoss creates a criterion to measure the average absolute error between x and y element-wise, where x is the input and y is the labels.	Ascend GPU CPU
mindspore.nn.MSELoss	Calculates the mean squared error between the predicted value and the label value.	Ascend GPU CPU
mindspore.nn.MultiClassDiceLoss	When there are multiple classifications, label is transformed into multiple binary classifications by one hot.	Ascend GPU CPU
mindspore.nn.MultilabelMarginLoss	Creates a loss criterion that minimizes the hinge loss for multi-class classification tasks.	Ascend GPU
mindspore.nn.MultiLabelSoftMarginLoss	Calculates the MultiLabelSoftMarginLoss.	Ascend GPU CPU
mindspore.nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 1D tensor of target class indices, $0 \leq y \leq$ x.size(1) $- 1$):	Ascend GPU CPU
mindspore.nn.NLLLoss	Gets the negative log likelihood loss between logits and labels.	Ascend GPU CPU
mindspore.nn.PoissonNLLLoss	Poisson negative log likelihood loss.	Ascend GPU CPU
mindspore.nn.RMSELoss	RMSELoss creates a criterion to measure the root mean square error between x and y element-wise, where x is the input and y is the labels.	Ascend GPU CPU
mindspore.nn.SampledSoftmaxLoss	Computes the sampled softmax training loss.	GPU
mindspore.nn.SmoothL1Loss	SmoothL1 loss function.	Ascend GPU CPU
mindspore.nn.SoftMarginLoss	A loss class for two-class classification problems.	Ascend GPU
mindspore.nn.SoftmaxCrossEntropyWithLogits	Computes softmax cross entropy between logits and labels.	Ascend GPU CPU
mindspore.nn.TripletMarginLoss	TripletMarginLoss operation.	GPU

3.14.1 mindspore.nn.BCELoss

class mindspore.nn.**BCELoss**(*weight=None*, *reduction='mean'*)

BCELoss creates a criterion to measure the binary cross entropy between the true labels and predicted labels.

Set the predicted labels as x, true labels as y, the output loss as $\ell(x, y)$. The formula is as follow:

$$L = \{l_1, \ldots, l_n, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

where N is the batch size. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Note: Note that the predicted labels should always be the output of sigmoid. Because it is a two-class classification, the true labels should be numbers between 0 and 1. And if input is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation.

Parameters

- **weight** (Tensor, *optional*) –A rescaling weight applied to the loss of each batch element. And it must have the same shape and data type as *inputs*. Default: None .
- **reduction** (*str*, *optional*) –Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the weighted mean of elements in the output.
  - 'sum': the output elements will be summed.

Inputs:

- **logits** (Tensor) - The input tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions. The data type must be float16 or float32.
- **labels** (Tensor) - The label tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions. The same shape and data type as *logits*.

Outputs:

Tensor, has the same dtype as *logits*. if *reduction* is 'none', then it has the same shape as *logits*. Otherwise, it is a scalar Tensor.

Raises

- **TypeError** –If dtype of *logits*, *labels* or *weight* (if given) is neither float16 not float32.
- **ValueError** –If *reduction* is not one of 'none', 'mean', 'sum'.
- **ValueError** –If shape of *logits* is not the same as *labels* or *weight* (if given).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> weight = ms.Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 3.3, 2.2]]), ms.float32)
>>> loss = nn.BCELoss(weight=weight, reduction='mean')
>>> logits = ms.Tensor(np.array([[0.1, 0.2, 0.3], [0.5, 0.7, 0.9]]), ms.float32)
>>> labels = ms.Tensor(np.array([[0, 1, 0], [0, 0, 1]]), ms.float32)
```

(continues on next page)

(continued from previous page)

```
>>> output = loss(logits, labels)
>>> print(output)
1.8952923
```

3.14.2 mindspore.nn.BCEWithLogitsLoss

class mindspore.nn.**BCEWithLogitsLoss**(*reduction='mean'*, *weight=None*, *pos_weight=None*)

Adds sigmoid activation function to input *input* as logits, and uses the given logits to compute binary cross entropy between the *input* and the *target*.

Sets input *input* as X, input *target* as Y, output as L. Then,

$$p_{ij} = sigmoid(X_{ij}) = \frac{1}{1 + e^{-X_{ij}}}$$

$$L_{ij} = -[Y_{ij} \cdot \log(p_{ij}) + (1 - Y_{ij}) \cdot \log(1 - p_{ij})]$$

Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **weight** (Tensor, *optional*) – A rescaling weight applied to the loss of each batch element. If not None, it can be broadcast to a tensor with shape of *input*, data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None`.
- **pos_weight** (Tensor, *optional*) – A weight of positive examples. Must be a vector with length equal to the number of classes. If not None, it must be broadcast to a tensor with shape of *input*, data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None`.

Inputs:

- **input** (Tensor) - Input *input* with shape $(N, *)$ where $*$ means, any number of additional dimensions. The data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **target** (Tensor) - Ground truth label with shape $(N, *)$ where $*$ means, any number of additional dimensions. The same shape and data type as *input*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, its shape is the same as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If input *input* or *target* is not Tensor.
- **TypeError** – If data type of *input* or *target* is not float16, float32 or bfloat16.

- **TypeError** –If *weight* or *pos_weight* is a parameter.
- **TypeError** –If data type of *weight* or *pos_weight* is not float16 , float32 or bfloat16.
- **TypeError** –If data type of *reduction* is not string.
- **ValueError** –If *weight* or *pos_weight* can not be broadcast to a tensor with shape of *input*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> input = ms.Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]).astype(np.float32))
>>> target = ms.Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]).astype(np.float32))
>>> loss = nn.BCEWithLogitsLoss()
>>> output = loss(input, target)
>>> print(output)
0.3463612
```

3.14.3 mindspore.nn.CosineEmbeddingLoss

class mindspore.nn.**CosineEmbeddingLoss**(*margin=0.0*, *reduction='mean'*)

CosineEmbeddingLoss creates a criterion to measure the similarity between two tensors using cosine distance.

Given two tensors $x1$, $x2$, and a Tensor label y with values 1 or -1:

$$loss(x_1, x_2, y) = \begin{cases} 1 - cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, cos(x_1, x_2) - margin), & \text{if } y = -1 \end{cases}$$

Parameters

- **margin** (*float*) –Should be in [-1.0, 1.0]. Default: `0.0` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits_x1** (Tensor) - Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **logits_x2** (Tensor) - Tensor of shape $(N, *)$, same shape and dtype as *logits_x1*.
- **labels** (Tensor) - Contains value 1 or -1. Suppose the shape of *logits_x1* is $(x_1, x_2, x_3, ..., x_R)$, then the shape of *labels* must be $(x_1, x_3, x_4, ..., x_R)$.

Outputs:

Tensor or Scalar, if *reduction* is `"none"`, its shape is the same as *labels*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If *margin* is not a float.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** – If *margin* is not in range [-1.0, 1.0].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> logits_x1 = ms.Tensor(np.array([[0.3, 0.8], [0.4, 0.3]]), ms.float32)
>>> logits_x2 = ms.Tensor(np.array([[0.4, 1.2], [-0.4, -0.9]]), ms.float32)
>>> labels = ms.Tensor(np.array([1, -1]), ms.int32)
>>> cosine_embedding_loss = nn.CosineEmbeddingLoss()
>>> output = cosine_embedding_loss(logits_x1, logits_x2, labels)
>>> print(output)
0.0003425479
```

3.14.4 mindspore.nn.CrossEntropyLoss

class mindspore.nn.**CrossEntropyLoss**(*weight=None*, *ignore_index=- 100*, *reduction='mean'*, *label_smoothing=0.0*)

The cross entropy loss between input and target.

The CrossEntropyLoss support two kind of targets:

- Class indices (int) in the range $[0, C)$ where C is the number of classes, when reduction is `none`, the loss can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}$$

  where x is the inputs, t is the target, w is the weight, N is the batch size, c belonging to [0, C-1] is class index, where C is the number of classes.

  If reduction is not `'none'` (default 'mean'), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

- Probabilities (float) for each class, useful when labels beyond a single class per minibatch item are required, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

  where x is the inputs, t is the target, w is the weight, N is the batch size, c belonging to [0, C-1] is class index, where C is the number of classes.

If reduction is not `'none'` (default 'mean'), then

$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **weight** (Tensor) –The rescaling weight to each class. If the value is not None, the shape is $(C,)$. The data type only supports float32 or float16. Default: `None` .
- **ignore_index** (*int*) –Specifies a target value that is ignored (typically for padding value) and does not contribute to the gradient. Default: `-100` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **label_smoothing** (*float*) –Label smoothing values, a regularization tool used to prevent the model from overfitting when calculating Loss. The value range is [0.0, 1.0]. Default value: `0.0` .

Inputs:

- **logits** (Tensor) - Tensor of shape $(C,)$ (N, C) or $(N, C, d_1, d_2, ..., d_K)$, where *C = number of classes*. Data type must be float16 or float32.
- **labels** (Tensor) - For class indices, tensor of shape (), (N) or $(N, d_1, d_2, ..., d_K)$, data type must be int32. For probabilities, tensor of shape $(C,)$ (N, C) or $(N, C, d_1, d_2, ..., d_K)$, data type must be float16 or float32.

Returns

Tensor, the computed cross entropy loss value.

Raises

- **TypeError** –If *weight* is not a Tensor.
- **TypeError** –If *ignore_index* is not an int.
- **TypeError** –If the data type of *weight* is not float16 or float32.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **TypeError** –If *label_smoothing* is not a float.
- **TypeError** –If *logits* is not a Tensor.
- **TypeError** –If *labels* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> # Case 1: Indices labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.array([1, 0, 4]), ms.int32)
>>> loss = nn.CrossEntropyLoss()
>>> output = loss(inputs, target)
>>> # Case 2: Probability labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> loss = nn.CrossEntropyLoss()
>>> output = loss(inputs, target)
```

3.14.5 mindspore.nn.CTCLoss

class mindspore.nn.**CTCLoss**(*blank=0*, *reduction='mean'*, *zero_infinity=False*)

Calculates the CTC (Connectionist Temporal Classification) loss. It's mainly used to calculate the loss between the continuous, unsegemented time series and the target series.

For the CTC algorithm, refer to Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks .

Parameters

- **blank** (*int, optional*) – The blank label. Default: `0` .
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **zero_infinity** (*bool, optional*) – If loss is infinite, this parameter determines whether to set that loss and its correlated gradient to zero. Default: `False` .

Inputs:

- **log_probs** (Tensor) - A tensor of shape (T, N, C) or (T, C), where T is length of input, N is size of the batch and C is the number of classes. T, N and C are positive integers.
- **targets** (Tensor) - A tensor of shape (N, S) or (sum(*target_lengths*)), where S is max target length, means the target sequences.
- **input_lengths** (Union[tuple, Tensor]) - A tuple or Tensor of shape (N). It means the lengths of the input.
- **target_lengths** (Union[tuple, Tensor]) - A tuple or Tensor of shape (N). It means the lengths of the target.

Outputs:

- **neg_log_likelihood** (Tensor) - A loss value which is differentiable with respect to each input node.

Raises

- **TypeError** – If *log_probs* or *targets* is not a Tensor.

- **TypeError** – If *zero_infinity* is not a bool, *reduction* is not string.
- **TypeError** – If the dtype of *log_probs* is not float or double.
- **TypeError** – If the dtype of *targets*, *input_lengths* or *target_lengths* is not int32 or int64.
- **ValueError** – If *reduction* is not `"none"`, `"mean"` or `"sum"`.
- **ValueError** – If the value of *blank* is not in range [0, C). C is number of classes of *log_probs* .
- **ValueError** – If the shape of *log_probs* is (T, C), the dimension of *targets* is not 1 or 2.
- **ValueError** – If the shape of *log_probs* is (T, C), the first dimension of 2-D *target* is not 1.
- **RuntimeError** – If any value of *input_lengths* is larger than T. T is length of *log_probs* .
- **RuntimeError** – If any target_lengths[i] is not in range [0, input_length[i]].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> T = 5      # Input sequence length
>>> C = 2      # Number of classes
>>> N = 2      # Batch size
>>> S = 3      # Target sequence length of longest target in batch (padding length)
>>> S_min = 2  # Minimum target length, for demonstration purposes
>>> arr = np.arange(T*N*C).reshape((T, N, C))
>>> ms_input = ms.Tensor(arr, dtype=ms.float32)
>>> input_lengths = np.full(shape=(N), fill_value=T)
>>> input_lengths = ms.Tensor(input_lengths, dtype=ms.int32)
>>> target_lengths = np.full(shape=(N), fill_value=S_min)
>>> target_lengths = ms.Tensor(target_lengths, dtype=ms.int32)
>>> target = np.random.randint(1, C, size=(N, S))
>>> target = ms.Tensor(target, dtype=ms.int32)
>>> ctc_loss = nn.CTCLoss(blank=0, reduction='none', zero_infinity=False)
>>> loss = ctc_loss(ms_input, target, input_lengths, target_lengths)
>>> print(loss)
[-45.79497  -55.794968]
>>> arr = np.arange(T*C).reshape((T, C))
>>> ms_input = ms.Tensor(arr, dtype=ms.float32)
>>> input_lengths = ms.Tensor([T], dtype=ms.int32)
>>> target_lengths = ms.Tensor([S_min], dtype=ms.int32)
>>> target = np.random.randint(1, C, size=(S_min,))
>>> target = ms.Tensor(target, dtype=ms.int32)
>>> ctc_loss = nn.CTCLoss(blank=0, reduction='none', zero_infinity=False)
>>> loss = ctc_loss(ms_input, target, input_lengths, target_lengths)
>>> print(loss)
-25.794968
```

3.14.6 mindspore.nn.DiceLoss

class mindspore.nn.**DiceLoss**(*smooth=1e-5*)

The Dice coefficient is a set similarity loss, which is used to calculate the similarity between two samples. The value of the Dice coefficient is 1 when the segmentation result is the best and is 0 when the segmentation result is the worst. The Dice coefficient indicates the ratio of the area between two objects to the total area. The function is shown as follows:

$$dice = 1 - \frac{2 * |pred \bigcap true|}{|pred| + |true| + smooth}$$

$pred$ represent *logits*, $true$ represent *labels* .

Parameters

smooth (*float, optional*) – A term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-5` .

Inputs:

- **logits** (Tensor) - Input predicted value. The data type must be float16 or float32.
- **labels** (Tensor) - Input target value. Same shape as the *logits*. The data type must be float16 or float32.

Outputs:

Tensor, a tensor of shape with the per-example sampled Dice losses.

Raises

- **ValueError** – If the dimension of *logits* is different from *labels*.
- **TypeError** – If the type of *logits* or *labels* is not a tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> loss = nn.DiceLoss(smooth=1e-5)
>>> logits = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]), mindspore.float32)
>>> labels = Tensor(np.array([[0, 1], [1, 0], [0, 1]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.38596618
```

3.14.7 mindspore.nn.FocalLoss

class mindspore.nn.**FocalLoss**(*weight=None*, *gamma=2.0*, *reduction='mean'*)

It is a loss function to solve the imbalance of categories and the difference of classification difficulty. The loss function proposed by Kaiming team in their paper Focal Loss for Dense Object Detection improves the effect of image object detection. The function is shown as follows:

$$FL(p_t) = -(1 - p_t)^{\gamma} \log(p_t)$$

Parameters

- **gamma** (*float*) – Gamma is used to adjust the steepness of weight curve in focal loss. Default: `2.0`.
- **weight** (*Union[Tensor, None]*) – A rescaling weight applied to the loss of each batch element. The dimension of weight should be 1. If None, no weight is applied. Default: `None`.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Tensor of shape should be (N, C) or (N, C, H) or (N, C, H, W). Where C is the number of classes. Its value is greater than 1. If the shape is (N, C, H, W) or (N, C, H), the H or product of H and W should be the same as labels.
- **labels** (Tensor) - Tensor of shape should be (N, C) or (N, C, H) or (N, C, H, W). The value of C is 1 or it needs to be the same as predict's C. If C is not 1, the shape of target should be the same as that of predict, where C is the number of classes. If the shape is (N, C, H, W) or (N, C, H), the H or product of H and W should be the same as logits. The value of *labels* is should be in the range [-C, C). Where C is the number of classes in logits.

Outputs:

Tensor or Scalar, if *reduction* is `"none"`, its shape is the same as *logits*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** – If the data type of *gamma* is not a float.
- **TypeError** – If *weight* is not a Tensor.
- **ValueError** – If *labels* dim is different from *logits*.
- **ValueError** – If *labels* channel is not 1 and *labels* shape is different from *logits*.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> logits = ms.Tensor([[0.8, 1.4], [0.5, 0.9], [1.2, 0.9]], ms.float32)
>>> labels = ms.Tensor([[1], [1], [0]], ms.int32)
>>> focalloss = nn.FocalLoss(weight=ms.Tensor([1, 2]), gamma=2.0, reduction='mean')
>>> output = focalloss(logits, labels)
>>> print(output)
0.12516622
```

3.14.8 mindspore.nn.GaussianNLLLoss

class mindspore.nn.**GaussianNLLLoss**(*, *full=False*, *eps=1e-6*, *reduction='mean'*)

Gaussian negative log likelihood loss.

The target values are considered to be samples from a Gaussian distribution, where the expectation and variance are predicted by a neural network. For *labels* modeled on a Gaussian distribution, *logits* to record expectations, and the variance *var* (elements are all positive), the calculated loss is:

$$\text{loss} = \frac{1}{2}\left(\log\left(\max\left(\text{var},\ \text{eps}\right)\right) + \frac{\left(\text{logits} - \text{labels}\right)^2}{\max\left(\text{var},\ \text{eps}\right)}\right) + \text{const.}$$

where eps is used for stability of log. When $full = True$, a constant will be added to the loss. If the shape of var and $logits$ are not the same (due to a homoscedastic assumption), their shapes must allow correct broadcasting.

Keyword Arguments

- **full** (bool, *optional*) – Whether include the constant term in the loss calculation. When $full = True$, the constant term *const.* will be $0.5 * log(2\pi)$. Default: False .
- **eps** (float, *optional*) – Used to improve the stability of log function. Default: 1e-6 .
- **reduction** (*str*, *optional*) – Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the mean of elements in the output.
  - 'sum': the output elements will be summed.

Inputs:

- **logits** (Tensor) - Tensor of shape $(N, *)$ or $(*)$ where $*$ means any number of additional dimensions.
- **labels** (Tensor) - Tensor of shape $(N, *)$ or $(*)$, same shape as the logits, or same shape as the logits but with one dimension equal to 1 (to allow for broadcasting).
- **var** - Tensor of shape $(N, *)$ or $(*)$, same shape as logits, or same shape as the logits but with one dimension equal to 1, or same shape as the logits but with one fewer dimension (to allow for broadcasting).

Returns

Tensor or Tensor scalar, the computed loss depending on $reduction$.

Raises

- **TypeError** – If *logits* is not a Tensor.
- **TypeError** – If *labels* is not a Tensor.

- **TypeError** – If *full* is not a bool.
- **TypeError** – If *eps* is not a float.
- **ValueError** – If *eps* is not a float within (0, inf).
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> arr1 = np.arange(8).reshape((4, 2))
>>> arr2 = np.array([2, 3, 1, 4, 6, 4, 4, 9]).reshape((4, 2))
>>> logits = ms.Tensor(arr1, ms.float32)
>>> labels = ms.Tensor(arr2, ms.float32)
>>> loss = nn.GaussianNLLLoss(reduction='mean')
>>> var = ms.Tensor(np.ones((4, 1)), ms.float32)
>>> output = loss(logits, labels, var)
>>> print(output)
1.4374993
```

Reference:

Nix, D. A. and Weigend, A. S., "Estimating the mean and variance of the target probability distribution", Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94), Orlando, FL, USA, 1994, pp. 55-60 vol.1, doi: 10.1109/ICNN.1994.374138.

3.14.9 mindspore.nn.HingeEmbeddingLoss

class mindspore.nn.**HingeEmbeddingLoss**(*margin=1.0*, *reduction='mean'*)

Calculate the Hinge Embedding Loss value based on the input 'logits' and' labels' (only including 1 or -1). Usually used to measure the similarity between two inputs.

The loss function for n-th sample in the mini-batch is

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

where $L = \{l_1, \ldots, l_N\}^\top$.

Parameters

- **margin** (*float, int, optional*) – Threshold defined by Hinge Embedding Loss *margin*. Represented as Δ in the formula. Default: `1.0`.

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - The predicted value, expressed as x in the equation. Tensor of shape $(*)$ where $*$ means any number of dimensions.
- **labels** (Tensor) - Label value, represented as y in the equation. Same shape as the logits, contains -1 or 1.

Returns

Tensor or Tensor scalar, the computed loss depending on $reduction$.

Raises

- **TypeError** –If *logits* is not a Tensor.
- **TypeError** –If *labels* is not a Tensor.
- **TypeError** –If *margin* is not a float or int.
- **ValueError** –If *labels* does not have the same shape as *logits* or they could not broadcast to each other.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> arr1 = np.array([0.9, -1.2, 2, 0.8, 3.9, 2, 1, 0, -1]).reshape((3, 3))
>>> arr2 = np.array([1, 1, -1, 1, -1, 1, -1, 1, 1]).reshape((3, 3))
>>> logits = ms.Tensor(arr1, ms.float32)
>>> labels = ms.Tensor(arr2, ms.float32)
>>> loss = nn.HingeEmbeddingLoss(reduction='mean')
>>> output = loss(logits, labels)
>>> print(output)
0.16666667
```

3.14.10 mindspore.nn.HuberLoss

class mindspore.nn.**HuberLoss**(*reduction='mean'*, *delta=1.0*)

HuberLoss calculate the error between the predicted value and the target value. It has the advantages of both L1Loss and MSELoss.

Assuming that the x and y are 1-D Tensor, length N, then calculate the loss of x and y without dimensionality reduction (the reduction parameter is set to "none"). The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top$$

with

$$l_n = \begin{cases} 0.5 * (x_n - y_n)^2, & \text{if } |x_n - y_n| < delta; \\ delta * (|x_n - y_n| - 0.5 * delta), & \text{otherwise.} \end{cases}$$

where N is the batch size. If *reduction* is not `"none"`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = "mean";} \\ \text{sum}(L), & \text{if reduction = "sum".} \end{cases}$$

Parameters

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **delta** (*Union[int, float]*) –The threshold to change between two type of loss. The value must be positive. Default: `1.0` .

Inputs:

- **logits** (Tensor) - Predicted value, Tensor of any dimension. The data type must be float16 or float32.
- **labels** (Tensor) - Target value, same dtype and shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor or Scalar, if *reduction* is `"none"`, return a Tensor with same shape and dtype as *logits*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If data type of *logits* or *labels* is neither float16 nor float32.
- **TypeError** –If data type of *logits* or *labels* are not the same.
- **TypeError** –If dtype of *delta* is neither float nor int.
- **ValueError** –If *delta* is less than or equal to 0.
- **ValueError** –If *reduction* is not one of `"none"`, `"mean"`, `"sum"`.
- **ValueError** –If *logits* and *labels* have different shapes and cannot be broadcasted to each other.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = nn.HuberLoss()
>>> logits = ms.Tensor(np.array([1, 2, 3]), ms.float32)
>>> labels = ms.Tensor(np.array([1, 2, 2]), ms.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.16666667
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = nn.HuberLoss(reduction="none")
>>> logits = ms.Tensor(np.array([1, 2, 3]), ms.float32)
>>> labels = ms.Tensor(np.array([[1, 1, 1], [1, 2, 2]]), ms.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0.  0.5 1.5]
 [0.  0.  0.5]]
```

3.14.11 mindspore.nn.KLDivLoss

class mindspore.nn.**KLDivLoss**(*reduction='mean'*)

Computes the Kullback-Leibler divergence between the *logits* and the *labels*.

For tensors of the same shape x and $target$, the updating formulas of KLDivLoss algorithm are as follows,

$$L(x, target) = target \cdot (\log target - \log x)$$

Then,

$$\ell(x, target) = \begin{cases} L(x, target), & \text{if reduction = 'none';} \\ \text{mean}(L(x, target)), & \text{if reduction = 'mean';} \\ \text{sum}(L(x, target))/x.\text{shape}[0], & \text{if reduction = 'batchmean';} \\ \text{sum}(L(x, target)), & \text{if reduction = 'sum'.} \end{cases}$$

where x represents *logits*, $target$ represents *labels*, and $\ell(x, target)$ represents *output*.

Note:

- Currently it does not support float64 input on *Ascend*.
- The output aligns with the mathematical definition of Kullback-Leibler divergence only when *reduction* is set to 'batchmean'.

Parameters

reduction (*str*) – Specifies the reduction to be applied to the output. Default: `'mean'` .

- On Ascend, the value of *reduction* must be one of `'batchmean'` , `'none'` or `'sum'` .
- On GPU, the value of *reduction* must be one of `'mean'` , `'none'` or `'sum'` .
- On CPU, the value of *reduction* must be one of `'mean'` , `'batchmean'` , `'none'` or `'sum'` .

Inputs:

- **logits** (Tensor) - The input Tensor. The data type must be float16, float32 or float64.
- **labels** (Tensor) - The label Tensor which has the same shape and data type as *logits*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, then output is a tensor and has the same shape as *logits*. Otherwise, it is a scalar.

Raises

- **TypeError** –If *reduction* is not a str.
- **TypeError** –If neither *logits* nor *labels* is a Tensor.
- **TypeError** –If dtype of *logits* or *labels* is not currently supported.
- **ValueError** –If shape of *logits* is not the same as *labels*.
- **RuntimeError** –If *logits* or *labels* is a scalar when *reduction* is 'batchmean'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> logits = ms.Tensor(np.array([0.2, 0.7, 0.1]), ms.float32)
>>> labels = ms.Tensor(np.array([0., 1., 0.]), ms.float32)
>>> loss = nn.KLDivLoss(reduction='mean')
>>> output = loss(logits, labels)
>>> print(output)
-0.23333333
```

3.14.12 mindspore.nn.L1Loss

class mindspore.nn.**L1Loss**(*reduction='mean'*)

L1Loss is used to calculate the mean absolute error between the predicted value and the target value.

Assuming that the x and y are 1-D Tensor, length N, then calculate the loss of x and y without dimensionality reduction (the reduction parameter is set to "none"). The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with } l_n = |x_n - y_n|,$$

where N is the batch size. If *reduction* is not `'none'`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

reduction (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.

- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Predicted value, Tensor of any dimension.
- **labels** (Tensor) - Target value, same shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor, data type is float.

Raises

- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** –If *logits* and *labels* have different shapes and cannot be broadcasted to each other.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = nn.L1Loss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.33333334
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = nn.L1Loss(reduction='none')
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0. 1. 2.]
 [0. 0. 1.]]
```

3.14.13 mindspore.nn.MarginRankingLoss

class mindspore.nn.**MarginRankingLoss**(*margin=0.0*, *reduction='mean'*)

MarginRankingLoss creates a criterion that measures the loss.

Given two tensors $input1$, $input2$ and a Tensor label $target$ with values 1 or -1, the operation is as follows:

$$\text{loss}(input1, input2, target) = \max(0, -target * (input1 - input2) + \text{margin})$$

Parameters

- **margin** (*float, optional*) –Specify the adjustment factor of the operation. Default: `0.0`.

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **input1** (Tensor) - Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **input2** (Tensor) - Tensor of shape $(N, *)$, same shape and dtype as *input1*.
- **target** (Tensor) - Contains value 1 or -1. Suppose the shape of *input1* is $(x_1, x_2, x_3, ..., x_R)$, then the shape of *target* must be $(x_1, x_2, x_3, ..., x_R)$.

Outputs:

Tensor or Scalar. if *reduction* is `'none'`, its shape is the same as *input1*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *margin* is not a float.
- **TypeError** –If *input1*, *input2* or *target* is not a Tensor.
- **TypeError** –If the types of *input1* and *input2* are inconsistent.
- **TypeError** –If the types of *input1* and *target* are inconsistent.
- **ValueError** –If the shape of *input1* and *input2* are inconsistent.
- **ValueError** –If the shape of *input1* and *target* are inconsistent.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` , `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> import numpy as np
>>> loss1 = nn.MarginRankingLoss(reduction='none')
>>> loss2 = nn.MarginRankingLoss(reduction='mean')
>>> loss3 = nn.MarginRankingLoss(reduction='sum')
>>> sign = ops.Sign()
>>> input1 = Tensor(np.array([0.3864, -2.4093, -1.4076]), ms.float32)
>>> input2 = Tensor(np.array([-0.6012, -1.6681, 1.2928]), ms.float32)
>>> target = sign(Tensor(np.array([-2, -2, 3]), ms.float32))
>>> output1 = loss1(input1, input2, target)
>>> print(output1)
[0.98759997 0.         2.7003999 ]
>>> output2 = loss2(input1, input2, target)
>>> print(output2)
1.2293333
>>> output3 = loss3(input1, input2, target)
```

(continues on next page)

(continued from previous page)

```
>>> print(output3)
3.6879997
```

3.14.14 mindspore.nn.MAELoss

class mindspore.nn.**MAELoss**(*reduction='mean'*)

MAELoss creates a criterion to measure the average absolute error between x and y element-wise, where x is the input and y is the labels.

For simplicity, let x and y be 1-dimensional Tensor with length N, the unreduced loss (i.e. with argument reduction set to 'none') of x and y is given as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with } l_n = |x_n - y_n|$$

where N is the batch size. If *reduction* is not `'none'`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Tensor of shape $(M, *)$ where $*$ means, any number of additional dimensions.
- **labels** (Tensor) - Tensor of shape $(N, *)$, same shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor, weighted loss float tensor, the shape is zero if *reduction* is `'mean'` or `'sum'` ., while the shape of output is the broadcasted shape if *reduction* is 'none'.

Raises

ValueError – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = nn.MAELoss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.33333334
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = nn.MAELoss(reduction='none')
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0. 1. 2.]
 [0. 0. 1.]]
```

3.14.15 mindspore.nn.MSELoss

class mindspore.nn.**MSELoss**(*reduction='mean'*)

Calculates the mean squared error between the predicted value and the label value.

For simplicity, let x and y be 1-dimensional Tensor with length N, the unreduced loss (i.e. with argument reduction set to 'none') of x and y is given as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with} \quad l_n = (x_n - y_n)^2.$$

where N is the batch size. If *reduction* is not `'none'`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - The predicted value of the input. Tensor of any dimension.
- **labels** (Tensor) - The input label. Tensor of any dimension, same shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor, loss of type float, the shape is zero if *reduction* is `'mean'` or `'sum'` , while the shape of output is the broadcasted shape if *reduction* is 'none'.

Raises

- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.
- **ValueError** –If *logits* and *labels* have different shapes and cannot be broadcasted.
- **TypeError** –if *logits* and *labels* have different data types.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = nn.MSELoss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 1, 1]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
1.6666667
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = nn.MSELoss(reduction='none')
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0. 1. 4.]
 [0. 0. 1.]]
```

3.14.16 mindspore.nn.MultiClassDiceLoss

class mindspore.nn.**MultiClassDiceLoss**(*weights=None*, *ignore_indiex=None*, *activation='softmax'*)

When there are multiple classifications, label is transformed into multiple binary classifications by one hot. For each channel section in the channel, it can be regarded as a binary classification problem, so it can be obtained through the binary *mindspore.nn.DiceLoss* losses of each category, and then the average value of the binary losses.

Parameters

- **weights** (*Union[Tensor, None]*) –Tensor of shape ($num\_classes, dim$). The weight shape[0] should be equal to labels shape[1]. Default: `None` .
- **ignore_indiex** (*Union[int, None]*) –Class index to ignore. Default: `None` .
- **activation** (*Union[str, Cell]*) –Activate function applied to the output of the fully connected layer, eg. 'ReLU'. Default: `'softmax'` . Choose from: [`'softmax'` , `'logsoftmax'` , `'relu'` , `'relu6'` , `'tanh'` , `'Sigmoid'`]

Inputs:

- **logits** (Tensor) - Tensor of shape $(N, C, *)$ where $*$ means, any number of additional dimensions. The logits dimension should be greater than 1. The data type must be float16 or float32.
- **labels** (Tensor) - Tensor of shape $(N, C, *)$, same shape as the *logits*. The labels dimension should be greater than 1. The data type must be float16 or float32.

Outputs:

Tensor, a tensor of shape with the per-example sampled MultiClass Dice Losses.

Raises

- **ValueError** –If the shape of *logits* is different from *labels*.
- **TypeError** –If the type of *logits* or *labels* is not a tensor.
- **ValueError** –If the dimension of *logits* or *labels* is less than 2.
- **ValueError** –If the weights.shape[0] is not equal to labels.shape[1].
- **ValueError** –If *weights* is a tensor, but its dimension is not 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> loss = nn.MultiClassDiceLoss(weights=None, ignore_indiex=None, activation="softmax")
>>> logits = Tensor(np.array([[0.2, 0.5, 0.7], [0.3, 0.1, 0.5], [0.9, 0.6, 0.3]]), mindspore.
↪float32)
>>> labels = Tensor(np.array([[0, 1, 0], [1, 0, 0], [0, 0, 1]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.54958105
```

3.14.17 mindspore.nn.MultilabelMarginLoss

class mindspore.nn.**MultilabelMarginLoss**(*reduction='mean'*)

Creates a loss criterion that minimizes the hinge loss for multi-class classification tasks. It takes a 2D mini-batch Tensor x as input and a 2D Tensor y containing target class indices as output.

Each sample in the mini-batch, the loss is computed as follows:

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$, $y \in \{0, \cdots, \text{y.size}(0) - 1\}$, $0 \leq y[j] \leq \text{x.size}(0) - 1$, and for all i and j, i does not equal to $y[j]$.

Furthermore, both y and x should have identical sizes.

Note: For this operator, only a contiguous sequence of non-negative targets that starts at the beginning is taken into consideration, which means that different samples can have different number of target classes.

Parameters

reduction (*str, optional*) –Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **x** (Tensor) - Predict data. Tensor of shape (C) or (N, C), where N is the batch size and C is the number of classes. Data type must be float16 or float32.
- **target** (Tensor) - Ground truth data, with the same shape as x, data type must be int32 and label targets padded by -1.

Outputs:

- **y** (Union[Tensor, Scalar]) - The loss of MultilabelMarginLoss. If *reduction* is `"none"`, its shape is (N). Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If x or *target* is not a Tensor.
- **TypeError** –If dtype of x is neither float16 nor float32.
- **TypeError** –If dtype of *target* is not int32.
- **ValueError** –If length of shape of x is neither 1 nor 2.
- **ValueError** –If shape of x is not the same as *target*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> loss = nn.MultilabelMarginLoss()
>>> x = ms.Tensor(np.array([[0.1, 0.2, 0.4, 0.8], [0.2, 0.3, 0.5, 0.7]]), ms.float32)
>>> target = ms.Tensor(np.array([[1, 2, 0, 3], [2, 3, -1, 1]]), ms.int32)
>>> output = loss(x, target)
>>> print(output)
0.325
```

3.14.18 mindspore.nn.MultiLabelSoftMarginLoss

class mindspore.nn.**MultiLabelSoftMarginLoss**(*weight=None*, *reduction='mean'*)

Calculates the MultiLabelSoftMarginLoss. The multi-label soft margin loss is a commonly used loss function in multi-label classification tasks where an input sample can belong to multiple classes. Given an input x and binary labels y of size (N, C), where N denotes the number of samples and C denotes the number of classes.

$$loss\,(x, y) = -\frac{1}{N}\frac{1}{C}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(y_{ij}\log\frac{1}{1+e^{-x_{ij}}} + (1-y_{ij})\log\frac{e^{-x_{ij}}}{1+e^{-x_{ij}}}\right)$$

where x_{ij} represents the predicted score of sample i for class j. y_{ij} represents the binary label of sample i for class j, where sample i belongs to class j if $y_{ij} = 1$, and sample i does not belong to class j if $y_{ij} = 0$. For a multi-label classification task, each sample may have multiple labels with a value of 1 in the binary label y. *weight* will multiply to the loss of each class if given.

Parameters

- **weight** (*Union[Tensor, int, float]*) –The manual rescaling weight given to each class. Default: `None` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **x** (Tensor) - A tensor of shape (N, C), where N is batch size and C is number of classes.
- **target** (Tensor) - The label target Tensor which has the same shape as x.

Outputs:

Tensor, the data type is the same as x, if the reduction is `'none'`, its shape is (N), otherwise it is zero.

Raises

ValueError –If the rank of x or *target* is not 2.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> x = ms.Tensor([[0.3, 0.6, 0.6], [0.9, 0.4, 0.2]])
>>> target = ms.Tensor([[0.0, 0.0, 1.0], [0.0, 0.0, 1.0]])
>>> loss = nn.MultiLabelSoftMarginLoss(reduction='mean')
>>> out = loss(x, target)
>>> print(out.asnumpy())
0.84693956
```

3.14.19 mindspore.nn.MultiMarginLoss

class mindspore.nn.**MultiMarginLoss**(*p=1*, *margin=1.0*, *reduction='mean'*, *weight=None*)

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 1D tensor of target class indices, $0 \leq y \leq \text{x.size}(1) - 1$):

For each mini-batch sample, the loss in terms of the 1D input x and scalar output y is:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, w[y] * (\text{margin} - x[y] + x[i]))^p}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$ and $i \neq y$.

Parameters

- **p** (*int, optional*) −The norm degree for pairwise distance. Should be 1 or 2. Default: `1` .
- **margin** (*float, optional*) −A parameter to change pairwise distance. Default: 1.0.
- **reduction** (*str, optional*) −Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **weight** (`Tensor`, *optional*) −The rescaling weight to each class with shape $(C,)$. Data type only support float32, float16 or float64. Default: `None` , all classes are weighted equally.

Inputs:

- **x** (Tensor) - Input x, with shape (N, C). Data type only supports float32, float16 or float64. x is *x* in the above formula.
- **target** (Tensor) - Ground truth labels, with shape $(N,)$. Data type only supports int64. The value of target should be non-negative, less than C. *target* is *y* in the above formula.

Outputs:

Tensor. When *reduction* is `'none'`, the shape is $(N,)$. Otherwise, it is a scalar. Has the same data type with *x*.

Raises

- **TypeError** −If dtype of *p* or *target* is not int.
- **TypeError** −If dtype of *margin* is not float.
- **TypeError** −If dtype of *reduction* is not str.
- **TypeError** −If dtype of *x* is not float16, float or float64.
- **TypeError** −If dtype of *weight* and *x* is not the same.
- **ValueError** −If *p* is not 1 or 2.
- **ValueError** −If *reduction* is not one of { `'none'` , `'sum'` , `'mean'` }.
- **ValueError** −If shape[0] of *x* is not equal to shape[0] of *target*.
- **ValueError** −If shape[1] of *x* is not equal to shape[0] of *weight*.
- **ValueError** −IF rank of *weight* is not 1.
- **ValueError** −If rank of *x* is not 2 or rank of 'target' is not 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> x = ms.Tensor(np.ones(shape=[3, 3]), ms.float32)
>>> target = ms.Tensor(np.array([1, 2, 1]), ms.int64)
>>> loss = nn.MultiMarginLoss()
>>> output = loss(x, target)
>>> print(output)
0.6666667
```

3.14.20 mindspore.nn.NLLLoss

class mindspore.nn.**NLLLoss**(*weight=None*, *ignore_index=- 100*, *reduction='mean'*)

Gets the negative log likelihood loss between logits and labels.

The nll loss with $reduction = none$ can be described as:

$$\ell(x, t) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{t_n} x_{n,t_n}, \quad w_c = \text{weight}\,[c] \cdot \mathbb{1}\{c \neq \text{ignore\_index}\}$$

where x is the logits, t is the labels, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If $reduction \neq none$ (default `'mean'`), then

$$\ell(x, t) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{tn}} l_n, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'} \end{cases}$$

Parameters

- **weight** (Tensor) – The rescaling weight to each class. If the value is not None, the shape is $(C,)$. The data type only supports float32 or float16. Default: `None`.
- **ignore_index** (*int*) – Specifies a target value that is ignored (typically for padding value) and does not contribute to the gradient. Default: `-100`.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Tensor of shape (N, C) or $(N, C, d_1, d_2, ..., d_K)$ for K-dimensional data, where $C = number\ of\ classes$. Data type must be float16 or float32. *inputs* needs to be logarithmic probability.
- **labels** (Tensor) -(N) or $(N, d_1, d_2, ..., d_K)$ for K-dimensional data. Data type must be int32.

Returns

Tensor, the computed negative log likelihood loss value.

Raises

- **TypeError** – If *weight* is not a Tensor.

- **TypeError** –If *ignore_index* is not an int.
- **TypeError** –If the data type of *weight* is not float16 or float32.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **TypeError** –If *logits* is not a Tensor.
- **TypeError** –If *labels* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> logits = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> labels = ms.Tensor(np.array([1, 0, 4]), ms.int32)
>>> loss = nn.NLLLoss()
>>> output = loss(logits, labels)
```

3.14.21 mindspore.nn.PoissonNLLLoss

class mindspore.nn.**PoissonNLLLoss**(*log_input=True*, *full=False*, *eps=1e-08*, *reduction='mean'*)

Poisson negative log likelihood loss.

The loss is:

$$\mathcal{L}_D = \sum_{i=0}^{|D|} (x_i - y_i \ln x_i + \ln y_i!)$$

where $\mathcal{L}_D$ is the loss, y_i is the *target*, x_i is the *input*.

If *log_input* is True, use $e^{x_i} - y_i x_i$ instead of $x_i - y_i \ln x_i$. When calculating logarithms, the lower bound of *input* is set to *eps* to avoid numerical errors.

If *full* is False, the last term $\ln y_i!$ will be omitted, otherwise the last term will be approximated using Stirling formula:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Note: Calculating the logarithm of a negative number or the exponent of a large positive number under Ascend will have a different range of return values and results different from those under GPU and CPU.

Parameters

- **log_input** (*bool, optional*) –Whether use log input. Default: `True` .
- **full** (*bool, optional*) –Whether include the Stirling approximation term in the loss calculation. Default: `False` .
- **eps** (*float, optional*) –Lower bound of *input* when calculating logarithms. Default: `1e-08` .

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **input** (Tensor) - The input Tensor. The shape can be any number of dimensions.
- **target** (Tensor) - The label Tensor which has the same shape as *input*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, then output is a tensor and has the same shape as *input*. Otherwise it is a scalar.

Raises

- **TypeError** –If *reduction* is not a str.
- **TypeError** –If neither *input* nor *target* is a tensor.
- **TypeError** –If dtype of *input* or *target* is not currently supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> x = ms.Tensor([[0.3, 0.7], [0.5, 0.5]])
>>> target = ms.Tensor([[1.0, 2.0], [3.0, 4.0]])
>>> loss = nn.PoissonNLLLoss()
>>> output = loss(x, target)
>>> print(output.asnumpy())
0.3652635
```

3.14.22 mindspore.nn.RMSELoss

class mindspore.nn.**RMSELoss**

RMSELoss creates a criterion to measure the root mean square error between x and y element-wise, where x is the input and y is the labels.

For simplicity, let x and y be 1-dimensional Tensor with length N, the loss of x and y is given as:

$$loss = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2}$$

Inputs:

- **logits** (Tensor) - Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **labels** (Tensor) - Tensor of shape $(N, *)$, same shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor, weighted loss float tensor and its shape is ().

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = nn.RMSELoss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.57735026
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = nn.RMSELoss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
1.0
```

3.14.23 mindspore.nn.SampledSoftmaxLoss

class mindspore.nn.**SampledSoftmaxLoss**(*num_sampled*, *num_classes*, *num_true=1*, *sampled_values=None*, *remove_accidental_hits=True*, *seed=0*, *reduction='none'*)

Computes the sampled softmax training loss. This operator can accelerate the training of the softmax classifier over a large number of classes. It is generally an underestimate of the full softmax loss.

Parameters

- **num_sampled** (*int*) –The number of classes to randomly sample per batch.
- **num_classes** (*int*) –The number of possible classes.
- **num_true** (*int*) –The number of labels classes per training example. Default: `1` .
- **sampled_values** (*Union[list, tuple]*) –List or tuple of (*sampled_candidates*, *true_expected_count*, *sampled_expected_count*) returned by a **CandidateSampler* function. Default to None, *UniformCandidateSampler* is applied. Default: `None` .
- **remove_accidental_hits** (*bool*) –Whether to remove "accidental hits" where a sampled class equals to one of the labels classes. Default: `True` .
- **seed** (*int*) –Random seed for candidate sampling. Default: `0`.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'none'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **weights** (Tensor) - The weights of input. Tensor of shape (C, dim).
- **bias** (Tensor) - Tensor of shape $(C,)$. The class biases.
- **labels** (Tensor) - Tensor of shape (N, num_true), type *int64, int32*. The labels classes.
- **logits** (Tensor) - Tensor of shape (N, dim). The forward activations of the input network.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, then output is a tensor with shape $(N,)$. Otherwise, the output is a scalar.

Raises

- **TypeError** –If *sampled_values* is not a list or tuple.
- **TypeError** –If dtype of *labels* is neither int32 nor int64.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** –If *num_sampled* or *num_true* is greater than *num_classes*.
- **ValueError** –If length of *sampled_values* is not equal to 3.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> mindspore.set_seed(1)
>>> loss = nn.SampledSoftmaxLoss(num_sampled=4, num_classes=7, num_true=1)
>>> weights = Tensor(np.random.randint(0, 9, [7, 10]), mindspore.float32)
>>> biases = Tensor(np.random.randint(0, 9, [7]), mindspore.float32)
>>> labels = Tensor([0, 1, 2])
>>> logits = Tensor(np.random.randint(0, 9, [3, 10]), mindspore.float32)
>>> output = loss(weights, biases, labels, logits)
>>> print(output)
[4.6051701e+01 1.4000047e+01 6.1989022e-06]
```

3.14.24 mindspore.nn.SmoothL1Loss

class mindspore.nn.**SmoothL1Loss**(*beta=1.0*, *reduction='none'*)

SmoothL1 loss function. Compare the error value element-wise and if the absolute error between the predicted value and the target value is less than the set threshold *beta*, the square term is used, otherwise the absolute error term is used.

Given two input x, y, the SmoothL1Loss can be described as follows:

$$L_i = \begin{cases} \frac{0.5(x_i - y_i)^2}{\text{beta}}, & \text{if } |x_i - y_i| < \text{beta} \\ |x_i - y_i| - 0.5 * \text{beta}, & \text{otherwise.} \end{cases}$$

Where beta represents the threshold *beta*.

If *reduction* is not *none*, then:

$$L = \begin{cases} \text{mean}(L_i), & \text{if reduction = 'mean';} \\ \text{sum}(L_i), & \text{if reduction = 'sum'.} \end{cases}$$

Note:

- SmoothL1Loss can be regarded as modified version of *mindspore.nn.L1Loss* or a combination of *mindspore.nn.L1Loss* and *mindspore.ops.L2Loss*.
- *mindspore.nn.L1Loss* computes the element-wise absolute difference between two input tensor while *mindspore.ops.L2Loss* computes the
- squared difference between two input tensors. *mindspore.ops.L2Loss* often leads to faster convergence but it is less robust to outliers, and the loss function has better robustness.

Parameters

- **beta** (*number, optional*) – The loss function calculates the threshold of the transformation between L1Loss and L2Loss. Default: `1.0`.
  - Ascend: The value should be equal to or greater than zero.
  - CPU/GPU: The value should be greater than zero.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'none'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Predictive value. Tensor of any dimension. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32, float64.
- **labels** (Tensor) - Ground truth data.
  - CPU/Ascend: has the same shape as the *logits*, *logits* and *labels* comply with the implicit type conversion rules to make the data types consistent.
  - GPU: has the same shape and dtype as the *logits*.

Outputs:

Tensor, if *reduction* is `'none'`, then output is a tensor with the same shape as *logits*. Otherwise the shape of output tensor is ().

Raises

- **TypeError** – If input *logits* or *labels* are not Tensor.
- **RuntimeError** – If dtype of *logits* or *labels* is not one of float16, float32, float64, bfloat16.
- **ValueError** – If shape of *logits* is not the same as *labels*.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **TypeError** – If *beta* is not a float, int or bool.

- **RuntimeError** –If *beta* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> loss = nn.SmoothL1Loss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[0.  0.  0.5]
```

3.14.25 mindspore.nn.SoftMarginLoss

class mindspore.nn.**SoftMarginLoss**(*reduction='mean'*)

A loss class for two-class classification problems.

SoftMarginLoss creates a criterion that optimizes a two-class classification logistic loss between input tensor x and labels tensor y (containing 1 or -1).

$$\text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{x.nelement()}$$

$x.nelement()$ represents the number of element of x .

Parameters

reduction (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Predict data. Data type must be float16, float32, bfloat16 (Among them, the Atlas training series products do not support bfloat16).
- **labels** (Tensor) - Ground truth data, with the same shape as *logits*. In GE mode, the data type should be the same as *logits*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, its shape is the same as *logits*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *logits* or *labels* is not a Tensor.

- **TypeError** –If dtype of *logits* or *labels* is not float16, float32, bfloat16 (Among them, the Atlas training series products do not support bfloat16).
- **ValueError** –If shape of *logits* is not the same as *labels*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> loss = nn.SoftMarginLoss()
>>> logits = Tensor(np.array([[0.3, 0.7], [0.5, 0.5]]), mindspore.float32)
>>> labels = Tensor(np.array([[-1, 1], [1, -1]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.6764238
```

3.14.26 mindspore.nn.SoftmaxCrossEntropyWithLogits

class mindspore.nn.**SoftmaxCrossEntropyWithLogits**(*sparse=False*, *reduction='none'*)

Computes softmax cross entropy between logits and labels.

Measures the distribution error between the probabilities of the input (computed with softmax function) and the labels where the classes are mutually exclusive (only one class is positive) using cross entropy loss.

Typical input into this function is unnormalized scores denoted as x whose shape is (N, C) , and the corresponding targets.

Typically, the input to this function is the fractional value of each category and the corresponding target value, and the input format is (N, C) .

For each instance x_i, i ranges from 0 to N-1, the loss is given as:

$$\ell(x_i, c) = -\log\left(\frac{\exp(x_i[c])}{\sum_j \exp(x_i[j])}\right) = -x_i[c] + \log\left(\sum_j \exp(x_i[j])\right)$$

where x_i is a 1D score Tensor, c is the index of 1 in one-hot.

Note: While the labels classes are mutually exclusive, i.e., only one class is positive in the labels, the predicted probabilities does not need to be exclusive. It is only required that the predicted probability distribution of entry is a valid one.

Parameters

- **sparse** (*bool, optional*) –Specifies whether labels use sparse format or not. Default: `False` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'none'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.

  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Tensor of shape (N, C) . Data type must be float16 or float32.
- **labels** (Tensor) - Tensor of shape $(N,)$. If *sparse* is True, The type of *labels* is int32 or int64. Otherwise, the type of *labels* is the same as the type of *logits*.

Outputs:

Tensor, a tensor of the same shape and type as logits with the component-wise logistic losses.

Raises

- **TypeError** –If *sparse* is not a bool.
- **TypeError** –If *sparse* is `True` and dtype of *labels* is neither int32 nor int64.
- **TypeError** –If *sparse* is `False` and dtype of *labels* is neither float16 not float32.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> # case 1: sparse=True
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> logits = Tensor(np.array([[3, 5, 6, 9, 12, 33, 42, 12, 32, 72]]), mindspore.float32)
>>> labels_np = np.array([1]).astype(np.int32)
>>> labels = Tensor(labels_np)
>>> output = loss(logits, labels)
>>> print(output)
[67.]
>>> # case 2: sparse=False
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=False)
>>> logits = Tensor(np.array([[3, 5, 6, 9, 12, 33, 42, 12, 32, 72]]), mindspore.float32)
>>> labels_np = np.array([[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]]).astype(np.float32)
>>> labels = Tensor(labels_np)
>>> output = loss(logits, labels)
>>> print(output)
[30.]
```

3.14.27 mindspore.nn.TripletMarginLoss

class mindspore.nn.**TripletMarginLoss**(*p=2*, *swap=False*, *eps=1e-06*, *reduction='mean'*, *margin=1.*)

TripletMarginLoss operation.

Triple loss is used to measure the relative similarity between samples, which is measured by a triplet and a *margin* with a value greater than 0 . The triplet is composed by a, p, n in the following formula.

The shapes of all input tensors should be $(N, *)$, where N is batch size and $*$ means any number of additional dimensions.

The distance swap is described in detail in the paper Learning local feature descriptors with triplets and shallow convolutional neural networks by V. Balntas, E. Riba et al.

The loss function for each sample in the mini-batch is:

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$$

where

$$d(x_i, y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_p$$

Parameters

- **p** (*int, optional*) –The degree of norm for pairwise distance. Default: `2` .
- **eps** (*float, optional*) –Add small value to avoid division by zero. Default: `1e-06` .
- **swap** (*bool, optional*) –The distance swap change the negative distance to the distance between positive sample and negative sample. Default: `False` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **margin** (*Union[Tensor, float]*) –Make a margin between the positive pair and the negative pair. The length of shape of *margin* must be 0. Default: `1.0` .

Inputs:

- **x** (Tensor) - A sample randomly selected from the training set. Data type must be BasicType. a in the above formula.
- **positive** (Tensor) - A sample belonging to the same category as x, with the same type and shape as x. p in the above formula.
- **negative** (Tensor) - A sample belonging to the different class from x, with the same type and shape as x. n in the above formula.
- **margin** (Union[Tensor, float]) - Make a margin between the positive pair and the negative pair. The length of shape of *margin* must be 0. Default: `1.0` .

Outputs:

Tensor. If *reduction* is `"none"`, its shape is (N). Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If x or *positive* or 'negative' is not a Tensor.
- **TypeError** –If dtype of x, *positive* and *negative* is not the same.

- **TypeError** – If *p* is not an int.
- **TypeError** – If *eps* is not a float.
- **TypeError** – If *swap* is not a bool.
- **ValueError** – If dimensions of input *x*, *positive* and *negative* are less than or equal to 1 at the same time.
- **ValueError** – If the dimension of input *x* or *positive* or *negative* is bigger than or equal to 8.
- **ValueError** – If length of shape of *margin* is not 0.
- **ValueError** – If shape of *x*, *positive* and *negative* cannot broadcast.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> import numpy as np
>>> loss = nn.TripletMarginLoss()
>>> x = ms.Tensor(np.array([[0.3, 0.7], [0.5, 0.5]]), ms.float32)
>>> positive = ms.Tensor(np.array([[0.4, 0.6], [0.4, 0.6]]), ms.float32)
>>> negative = ms.Tensor(np.array([[0.2, 0.9], [0.3, 0.7]]), ms.float32)
>>> output = loss(x, positive, negative)
>>> print(output)
0.8881968
```

3.15 Optimizer

API Name	Description	Supported Platforms
mindspore.nn.Adadelta	Implements the Adadelta algorithm.	`Ascend` `GPU` `CPU`
mindspore.nn.Adagrad	Implements the Adagrad algorithm.	`Ascend` `GPU` `CPU`
mindspore.nn.Adam	Implements the Adaptive Moment Estimation (Adam) algorithm.	`Ascend` `GPU` `CPU`
mindspore.nn.AdaMax	Implements the AdaMax algorithm, a variant of Adaptive Movement Estimation (Adam) based on the infinity norm.	`Ascend` `GPU` `CPU`
mindspore.nn.AdamWeightDecay	Implements the Adam algorithm with weight decay.	`Ascend` `GPU` `CPU`
mindspore.nn.AdaSumByDeltaWeightWrapCell	Enable the adasum in "auto_parallel/semi_auto_parallel" mode.	`Ascend` `GPU`
mindspore.nn.AdaSumByGradWrapCell	Enable the adasum in "auto_parallel/semi_auto_parallel" mode.	`Ascend` `GPU`
mindspore.nn.ASGD	Implements Average Stochastic Gradient Descent.	`Ascend` `GPU` `CPU`
mindspore.nn.FTRL	Implements the FTRL algorithm.	`Ascend` `GPU`
mindspore.nn.Lamb	Implements the Lamb(Layer-wise Adaptive Moments optimizer for Batching training) algorithm.	`Ascend` `GPU`
mindspore.nn.LARS	Implements the LARS algorithm.	`Ascend`

continues on next page

Table 15 – continued from previous page

mindspore.nn.LazyAdam	Implements the Adaptive Moment Estimation (Adam) algorithm.	Ascend GPU CPU
mindspore.nn.Momentum	Implements the Momentum algorithm.	Ascend GPU CPU
mindspore.nn.OptTFTWrapper	Implements TFT optimizer wrapper, this wrapper is used to report status to MindIO TFT before optimizer updating.	Ascend
mindspore.nn.ProximalAdagrad	Implements the ProximalAdagrad algorithm that is an online Learning and Stochastic Optimization.	Ascend GPU
mindspore.nn.RMSProp	Implements Root Mean Squared Propagation (RMSProp) algorithm.	Ascend GPU CPU
mindspore.nn.Rprop	Implements Resilient backpropagation.	Ascend GPU CPU
mindspore.nn.SGD	Implements stochastic gradient descent.	Ascend GPU CPU

3.15.1 mindspore.nn.Adadelta

class mindspore.nn.**Adadelta**(*params*, *learning_rate=1.0*, *rho=0.9*, *epsilon=1e-6*, *loss_scale=1.0*, *weight_decay=0.0*)

Implements the Adadelta algorithm.

Adadelta is an online Learning and Stochastic Optimization. Refer to paper ADADELTA: AN ADAPTIVE LEARNING RATE METHOD.

$$
\begin{aligned}
accum_t &= \rho * accum_{t-1} + (1-\rho) * g_t^2 \\
update_t &= \sqrt{accum\_update_{t-1} + \epsilon} * \frac{g_t}{\sqrt{accum_t + \epsilon}} \\
accum\_update_t &= \rho * accum\_update_{t-1} + (1-\rho) * update_t^2 \\
w_t &= w_{t-1} - \gamma * update_t
\end{aligned}
$$

where g represents *grads*, γ represents *learning_rate*, ρ represents *rho*, ϵ represents *epsilon*, w represents *params*, $accum$ represents accumulation, $accum\_update$ represents accumulation update, t represents current step.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string *"params"*, *"lr"*, *"weight_decay"*, *"grad_centralization"* and *"order_params"* are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is `False` by default. This configuration only works on the convolution layer.

  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.

- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `1.0`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **rho** (*float, optional*) – Decay rate, must be in range [0.0, 1.0]. Default: `0.9`.
- **epsilon** (*float, optional*) – A small value added for numerical stability, must be non-negative. Default: `1e-6`.
- **loss_scale** (*float, optional*) – Value for the loss scale. It must be greater than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **grads** (tuple[Tensor]) - The gradients of *params* in the optimizer, has the same shape and data type as the *params* in optimizer. With float16 or float32 data type.

Outputs:

Tensor[bool], the value is `True`.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *params* is neither Parameter nor dict.
- **TypeError** – If *rho*, *epsilon* or *loss_scale* is not a float.
- **TypeError** – If *weight_decay* is not float, int or cell.
- **ValueError** – if *rho* is not in range [0.0, 1.0].
- **ValueError** – If *loss_scale* is less than or equal to 0.
- **ValueError** – If *learning_rate*, *epsilon* or *weight_decay* is less than 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.conv = nn.Conv1d(120, 240, 4)
...         self.fc = nn.Dense(4, 1)
...     def construct(self, x):
...          x = self.conv(x)
...          x = self.fc(x)
...          return x
>>> net = Net()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Adadelta(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.Adadelta(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.2 mindspore.nn.Adagrad

class mindspore.nn.**Adagrad**(*params*, *accum=0.1*, *learning_rate=0.001*, *update_slots=True*, *loss_scale=1.0*, *weight_decay=0.0*)

Implements the Adagrad algorithm.

Adagrad is an online Learning and Stochastic Optimization. Refer to paper Efficient Learning using Forward-Backward Splitting. Adagrad can adaptively assign different learning rates to each parameter in response to the uneven number of samples for different parameters.

The updating Pseudo codes are as follows:

Parameters : learning rate γ, params w_0, weight decay λ,
initial accumulator value $state\_sum$
Init : $state\_sum_0 \leftarrow 0$

---

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_w f_t(w_{t-1})$
if $\lambda \neq 0$
$g_t \leftarrow g_t + \lambda w_{t-1}$
$state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$
$w_t \leftarrow w_{t-1} - \gamma * \frac{g_t}{\sqrt{state\_sum_t}+\epsilon}$

---

return $\mathbf{w_t}$

---

state_sum stands for the accumulated squared sum of the gradients $accum$. g stands for *grads*, λ stands for *weight_decay*. γ stands for *learning_rate*, w stands for *params*. t represents current *step*.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **accum** (*float, optional*) – The starting value for h, must be zero or positive values. Default: `0.1`.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `0.001`.

  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.

- **update_slots** (*bool, optional*) – Whether the *h* will be updated. Default: `True` .
- **loss_scale** (*float, optional*) – Value for the loss scale. It must be greater than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to False, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0` .
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0` .
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **grads** (tuple[Tensor]) - The gradients of *params* in the optimizer, the shape is the same as the *params* in optimizer.

Outputs:

Tensor[bool], the value is `True` .

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *accum* or *loss_scale* is not a float.
- **TypeError** – If *update_slots* is not a bool.
- **TypeError** – If *weight_decay* is neither float nor int.
- **ValueError** – If *loss_scale* is less than or equal to 0.
- **ValueError** – If *accum* or *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import train
>>> import mindspore.nn as nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Adagrad(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.Adagrad(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.3 mindspore.nn.Adam

class mindspore.nn.**Adam**(*params*, *learning_rate=1e-3*, *beta1=0.9*, *beta2=0.999*, *eps=1e-8*, *use_locking=False*, *use_nesterov=False*, *weight_decay=0.0*, *loss_scale=1.0*, *use_amsgrad=False*, ***kwargs*)

Implements the Adaptive Moment Estimation (Adam) algorithm.

The Adam optimizer can dynamically adjust the learning rate of each parameter using the first-order moment estimation and the second-order moment estimation of the gradient. The Adam algorithm is proposed in Adam: A Method for Stochastic Optimization.

The updating formulas are as follows:

Parameters : 1^{st} moment vector m, 2^{nd} moment vector v,

gradients g, learning rate γ, exponential decay rates for the moment estimates β_1 β_2,

parameter vector w_0, timestep t, weight decay λ

Init : $m_0 \leftarrow 0$, $v_0 \leftarrow 0$, $t \leftarrow 0$, init parameter vector w_0

---

while w_t not converged **do**

$\boldsymbol{g}_t \leftarrow \nabla_w \boldsymbol{f}_t\left(\boldsymbol{w}_{t-1}\right)$

if $\lambda \neq 0$

$\boldsymbol{g}_t \leftarrow \boldsymbol{g}_t + \lambda \boldsymbol{w}_{t-1}$

$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\, \boldsymbol{g}_t$

$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\, \boldsymbol{g}_t^2$

$\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / \left(1-\beta_1^t\right)$

$\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / \left(1-\beta_2^t\right)$

$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \gamma \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)$

end while

---

return $\boldsymbol{w}_t$

---

m represents the 1st moment vector, v represents the 2nd moment vector, g represents *gradients*, β_1, β_2 represent *beta1* and *beta2*, t represents the current step while $beta_1^t$ and $beta_2^t$ represent *beta1_power* and *beta2_power*, γ represents *learning_rate*, w represents *params*, ϵ represents *eps*.

Note: On Ascend, when *use_amsgrad* is set to True, it might have slightly larger accuracy error.

The sparse strategy is applied while the SparseGatherV2 operator is used for forward network. If the sparse strategy wants to be executed on the host, set the target to the CPU. The sparse feature is under continuous development.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

When using Adam with use_lazy=True:

Please note, the optimizer only updates the current index position of the network parameters when the gradient is sparse. The sparse behavior is not equivalent to the original Adam algorithm. If you want to execute a sparse policy, target needs to be set to CPU.

When using Adam with use_offload=True:

This optimizer only supports *GRAPH_MODE* and don't support GE backend.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) –Default: `1e-3`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **beta1** (*float, optional*) –The exponential decay rate for the 1st moment estimations. Should be in range (0.0, 1.0). Default: `0.9`.
- **beta2** (*float, optional*) –The exponential decay rate for the 2nd moment estimations. Should be in range (0.0, 1.0). Default: `0.999`.
- **eps** (*float, optional*) –Term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-8`.
- **use_locking** (*bool, optional*) –Whether to enable a lock to protect the updating process of variable tensors. If `true`, updates of the *w*, *m*, and *v* tensors will be protected by a lock. If `false`, the result is unpredictable. Default: `False`.

- **use_nesterov** (*bool, optional*) – Whether to use Nesterov Accelerated Gradient (NAG) algorithm to update the gradients. If `true` , update the gradients using NAG. If `false` , update the gradients without using NAG. Default: `False` .
- **use_amsgrad** (*bool, optional*) – Whether to use Amsgrad algorithm to update the gradients. If `true` , update the gradients using Amsgrad. If `false` , update the gradients without using Amsgrad. Default: `False` .
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0` .
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.
- **loss_scale** (*float, optional*) – A floating point value for the loss scale. Should be greater than 0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to False, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **kwargs** –
  - use_lazy (bool): Whether to use Lazy Adam algorithm. Default: `False` . If `true` , apply lazy adam algorithm. If `false` , apply normal adam algorithm.
  - use_offload (bool): Whether to offload adam optimizer to host CPU and keep parameters being updated on the device in order to minimize the memory cost. Default: `False` . If `true` , apply offload adam. If `false` , apply normal adam.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is True.

Raises

- **KeyError** – If kwargs got keys other than 'use_lazy' or 'use_offload'.
- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *params* is neither Parameter nor dict.
- **TypeError** – If *beta1*, *beta2*, *eps* or *loss_scale* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **TypeError** – If *use_locking*, *use_nesterov*, *use_amsgrad*, *use_lazy* or *use_offload* is not a bool.
- **ValueError** – If *loss_scale* or *eps* is less than or equal to 0.
- **ValueError** – If *beta1*, *beta2* is not in range (0.0, 1.0).
- **ValueError** – If *weight_decay* is less than 0.
- **ValueError** – If *use_lazy* and *use_offload* are both `true` .
- **ValueError** – If *use_amsgrad* is `true`, *use_lazy* or *use_offload* is `true` .
- **ValueError** – If *use_amsgrad* is `True` while using distributed training.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Adam(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.Adam(group_params, learning_rate=0.1, weight_decay=0.0, use_lazy=False, use_
 ↪offload=False)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.4 mindspore.nn.AdaMax

class mindspore.nn.**AdaMax**(*params*, *learning_rate=0.001*, *beta1=0.9*, *beta2=0.999*, *eps=1e-08*, *weight_decay=0.0*, *loss_scale=1.0*)

Implements the AdaMax algorithm, a variant of Adaptive Movement Estimation (Adam) based on the infinity norm.

The AdaMax algorithm is proposed in Adam: A Method for Stochastic Optimization.

The updating formulas are as follows,

$$
\begin{aligned}
m_{t+1} &= \beta_1 * m_t + (1 - \beta_1) * g \\
v_{t+1} &= \max(\beta_2 * v_t, |g|) \\
w &= w - \frac{l}{1-\beta_1^{t+1}} * \frac{m_{t+1}}{v_{t+1}+\epsilon}
\end{aligned}
$$

m represents the 1st moment vector, v represents the 2nd moment vector, g represents *gradients*, β_1, β_2 represent *beta1* and *beta2*, t represents the current step, $beta_1^t$ represent *beta1_power*, l represents *learning_rate*, w represents *params*, ϵ represents *eps*.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped,

each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule]*) –Default: `0.001`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **beta1** (*float, optional*) –The exponential decay rate for the 1st moment estimations. Should be in range (0.0, 1.0). Default: `0.9`.
- **beta2** (*float, optional*) –The exponential decay rate for the 2nd moment estimations. Should be in range (0.0, 1.0). Default: `0.999`.
- **eps** (*float, optional*) –Term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-08`.
- **weight_decay** (*Union[float, int, Cell]*) –Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

- **loss_scale** (*float, optional*) – A floating point value for the loss scale. Should be greater than 0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False` , then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0` .

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is True.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *beta1*, *beta2*, *eps* or *loss_scale* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **ValueError** – If *loss_scale* or *eps* is less than or equal to 0.
- **ValueError** – If *beta1*, *beta2* is not in range (0.0, 1.0).
- **ValueError** – If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.AdaMax(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.AdaMax(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
```

(continues on next page)

(continued from previous page)

```
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.5 mindspore.nn.AdamWeightDecay

class mindspore.nn.**AdamWeightDecay**(*params*, *learning_rate=1e-3*, *beta1=0.9*, *beta2=0.999*, *eps=1e-6*, *weight_decay=0.0*)

Implements the Adam algorithm with weight decay.

Parameters : 1^{st} moment vector m, 2^{nd} moment vector v,

$gradients\ g$, learning rate γ, exponential decay rates for the moment estimates β_1 β_2,

parameter vector w_0, timestep t, weight decay λ

Init : $m_0 \leftarrow 0$, $v_0 \leftarrow 0$, $t \leftarrow 0$, init parameter vector w_0

---

repeat

$t \leftarrow t + 1$

$\boldsymbol{g}_t \leftarrow \nabla f_t\left(\boldsymbol{w}_{t-1}\right)$

$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + \left(1 - \beta_1\right) \boldsymbol{g}_t$

$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + \left(1 - \beta_2\right) \boldsymbol{g}_t^2$

$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \gamma \left(\boldsymbol{m}_t / \left(\sqrt{\boldsymbol{v}_t} + \epsilon\right) + \lambda \boldsymbol{w}_{t-1}\right)$

until stopping criterion is met

---

return $\boldsymbol{w}_t$

---

m represents the 1st moment vector *moment1*, v represents the 2nd moment vector *moment2*, g represents *gradients*, γ represents *learning_rate*, β_1, β_2 represent *beta1* and *beta2*, t represents the current step, w represents *params*, λ represents *weight_decay*.

Note: There is usually no connection between a optimizer and mixed precision. But when *FixedLossScaleManager* is used and *drop_overflow_update* in *FixedLossScaleManager* is set to False, optimizer needs to set the 'loss_scale'. As this optimizer has no argument of *loss_scale*, so *loss_scale* needs to be processed by other means, refer document LossScale to process *loss_scale* correctly.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule]*) –Default: `1e-3`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **beta1** (*float*) –The exponential decay rate for the 1st moment estimations. Default: `0.9`. Should be in range (0.0, 1.0).
- **beta2** (*float*) –The exponential decay rate for the 2nd moment estimations. Default: `0.999`. Should be in range (0.0, 1.0).
- **eps** (*float*) –Term added to the denominator to improve numerical stability. Default: `1e-6`. Should be greater than 0.
- **weight_decay** (*Union[float, int, Cell]*) –Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

tuple[bool], all elements are True.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *beta1*, *beta2* or *eps* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **ValueError** – If *eps* is less than or equal to 0.
- **ValueError** – If *beta1*, *beta2* is not in range (0.0, 1.0).
- **ValueError** – If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.AdamWeightDecay(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01},
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.AdamWeightDecay(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
↪0.01.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
↪decay of 0.0.
>>> # The final parameters order in which the optimizer will be followed is the value of
↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.6 mindspore.nn.AdaSumByDeltaWeightWrapCell

class mindspore.nn.**AdaSumByDeltaWeightWrapCell**(*optimizer*)

Enable the adasum in "auto_parallel/semi_auto_parallel" mode. The implementation of the Adaptive Summation (AdaSum) algorithm is calculated based on the difference of weights before and after the updating of optimizer. See the paper AdaSum: Scaling Distributed Training with Adaptive Summation.

$$w_{t+1} = w_t - \alpha \cdot Adasum(g_1, g_2)$$
$$w_{t+1} = w_t - \alpha \cdot [(1 - \frac{g_2^T \cdot g_1}{2 \cdot \|g_1\|^2}) \cdot g_1 + (1 - \frac{g_1^T \cdot g_2}{2 \cdot \|g_2\|^2}) \cdot g_2]$$

In this implementation, g represents the weight difference before and after the updating of optimizer, and the subscripts represent different devices in the data parallel dimension.

Note: When using AdaSum, the number of traning cards needs to be a power of 2 and at least 16 cards are required. Currently, the optimizer sharding and pipeline parallel is not supported when using AdaSum. It is recommended to using AdaSumByDeltaWeightWrapCell in semi auto parallel/auto parallel mode. In data parallel mode, we recommend to using mindspore.boost to applying AdaSum.

Parameters

optimizer (*Union[Cell]*) – Optimizer for updating the weights. The construct function of the optimizer requires only one input.

Inputs:

- **grads** (Tuple(Tensor)) - Tuple of gradients, same with the input of passed optimizer.

Raises

- **RuntimeError** – If *parallel_mode* uses *stand_alone* mode, AdaSum only supports use in distributed scenarios.
- **RuntimeError** – If the optimizer parallel is used when using AdaSum.
- **RuntimeError** – If the pipeline parallel is used when using AdaSum.
- **RuntimeError** – If *device_num* is not a power of 2, or less than 16.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> optim = nn.AdaSumByDeltaWeightWrapCell(nn.Momentum(params=net.trainable_params(),
...                                                    learning_rate=0.1, momentum=0.9))
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim, metrics=None)
```

3.15.7 mindspore.nn.AdaSumByGradWrapCell

class mindspore.nn.**AdaSumByGradWrapCell**(*optimizer*)

Enable the adasum in "auto_parallel/semi_auto_parallel" mode. The implementation of the Adaptive Summation (AdaSum) algorithm is calculated by gradients. See the paper AdaSum: Scaling Distributed Training with Adaptive Summation.

$$w_{t+1} = w_t - \alpha \cdot Adasum(g_1, g_2)$$
$$w_{t+1} = w_t - \alpha \cdot [(1 - \frac{g_2^T \cdot g_1}{2 \cdot \|g_1\|^2}) \cdot g_1 + (1 - \frac{g_1^T \cdot g_2}{2 \cdot \|g_2\|^2}) \cdot g_2]$$

In this implementation, g represents the gradient of the weights, and the subscripts represent different devices in the data-parallel dimension.

Note:

- It is recommended to using AdaSumByGradWrapCell in semi auto parallel/auto parallel mode. In data parallel mode, we recommend to using mindspore.boost to applying AdaSum.
- When using AdaSum, the number of traning cards needs to be a power of 2 and at least 16 cards are required. Currently, the optimizer sharding and pipeline parallel is not supported when using AdaSum.

Parameters

optimizer (*Union[Cell]*) – Optimizer for updating the weights. The construct function of the optimizer requires only one input.

Inputs:

- **grads** (Tuple[Tensor]) - Tuple of gradients, same with the input of passed optimizer.

Raises

- **RuntimeError** – If *parallel_mode* uses *stand_alone* mode, AdaSum only supports use in distributed scenarios.
- **RuntimeError** – If the optimizer parallel is used when using AdaSum.
- **RuntimeError** – If the pipeline parallel is used when using AdaSum.
- **RuntimeError** – If *device_num* is not a power of 2, or less than 16.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> optim = nn.AdaSumByGradWrapCell(nn.Momentum(params=net.trainable_params(), learning_
↪rate=0.1, momentum=0.9))
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim, metrics=None)
```

3.15.8 mindspore.nn.ASGD

class mindspore.nn.**ASGD**(*params*, *learning_rate=0.1*, *lambd=1e-4*, *alpha=0.75*, *t0=1e6*, *weight_decay=0.0*)

Implements Average Stochastic Gradient Descent.

Introduction to ASGD can be found at Acceleration of stochastic approximation by average.

The updating formulas are as follows:

$$
\begin{aligned}
w_t &= w_{t-1} * (1 - \lambda * \eta_{t-1}) - \eta_{t-1} * g_t \\
ax_t &= (w_t - ax_{t-1}) * \mu_{t-1} \\
\eta_t &= \frac{1.}{(1+\lambda * lr * t)^{\alpha}} \\
\mu_t &= \frac{1}{\max(1, t-t0)}
\end{aligned}
$$

λ represents the decay term, μ and η are tracked to update ax and w, $t0$ represents the point of starting averaging, α represents the power for η update, ax represents the averaged parameter value, t represents the current step, g represents *gradients*, w represents *params*.

Note: This optimizer don't support GE backend.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*, if not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *parameters* is a list of *dict*, the "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule]*) – learning_rate. Default: 0.1 .
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.

  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.

- **lambd** (*float*) –The decay term. Default: `1e-4` .
- **alpha** (*float*) –The power for η update. Default: `0.75` .
- **t0** (*float*) –The point of starting averaging. Default: `1e6` .
- **weight_decay** (*Union[float, int, Cell]*) –Weight decay (L2 penalty). Default: `0.0` .
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is True.

Raises

- **TypeError** –If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** –If element of *parameters* is neither Parameter nor dict.
- **TypeError** –If *lambd*, *alpha* or *t0* is not a float.
- **TypeError** –If *weight_decay* is neither float nor int.
- **ValueError** –If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.ASGD(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params,'grad_centralization':True},
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.ASGD(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 default weight␣
↪decay of 0.0 and grad
```

(continues on next page)

(continued from previous page)

```
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.9 mindspore.nn.FTRL

class mindspore.nn.**FTRL**(*params*, *initial_accum=0.1*, *learning_rate=0.001*, *lr_power=- 0.5*, *l1=0.0*, *l2=0.0*, *use_locking=False*, *loss_scale=1.0*, *weight_decay=0.0*)

Implements the FTRL algorithm.

FTRL is an online convex optimization algorithm that adaptively chooses its regularization function based on the loss functions. Refer to paper Adaptive Bound Optimization for Online Convex Optimization.

The updating formulas are as follows,

$$
\begin{aligned}
m_{t+1} &= m_t + g^2 \\
u_{t+1} &= u_t + g - \frac{m_{t+1}^{-p} - m_t^{-p}}{\alpha} * \omega_t \\
\omega_{t+1} &= \begin{cases} \frac{(sign(u_{t+1}) * l1 - u_{t+1})}{\frac{m_{t+1}^{-p}}{\alpha} + 2 * l2} & \text{if } |u_{t+1}| > l1 \\ 0.0 & \text{otherwise} \end{cases}
\end{aligned}
$$

m represents accumulators, g represents *grads*, t represents the current step, u represents the linear coefficient to be updated, p represents *lr_power*, α represents *learning_rate*, ω represents *params*.

Note: The sparse strategy is applied while the SparseGatherV2 operator is used for forward network. If the sparse strategy wants to be executed on the host, set the target to the CPU. The sparse feature is under continuous development.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Using different learning rate by grouping parameters is currently not supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.

  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.

- **initial_accum** (*float, optional*) – The starting value for accumulators *m*, must be zero or positive values. Default: `0.1`.
- **learning_rate** (*float, optional*) – The learning rate value, must be zero or positive, dynamic learning rate is currently not supported. Default: `0.001`.
- **lr_power** (*float, optional*) – Learning rate power controls how the learning rate decreases during training, must be less than or equal to zero. Use fixed learning rate if lr_power is zero. Default: `-0.5`.
- **l1** (*float, optional*) – l1 regularization strength, must be greater than or equal to zero. Default: `0.0`.
- **l2** (*float, optional*) – l2 regularization strength, must be greater than or equal to zero. Default: `0.0`.
- **use_locking** (*bool, optional*) – If true, use locks for updating operation. Default: `False`.
- **loss_scale** (*float, optional*) – Value for the loss scale. It must be greater than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **grads** (tuple[Tensor]) - The gradients of *params* in the optimizer, the shape is the same as the *params* in optimizer.

Outputs:

Tuple[Parameter], the updated parameters, the shape is the same as *params*.

Raises

- **TypeError** – If *initial_accum*, *learning_rate*, *lr_power*, *l1*, *l2* or *loss_scale* is not a float.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *weight_decay* is neither float nor int.
- **TypeError** – If *use_nesterov* is not a bool.
- **ValueError** – If *lr_power* is greater than 0.
- **ValueError** – If *loss_scale* is less than or equal to 0.
- **ValueError** – If *initial_accum*, *l1* or *l2* is less than 0.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.FTRL(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.FTRL(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use default learning rate of 0.1 will use default␣
 ↪weight decay
>>> # of 0.0 and grad centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.10 mindspore.nn.Lamb

class mindspore.nn.**Lamb**(*params*, *learning_rate*, *beta1=0.9*, *beta2=0.999*, *eps=1e-6*, *weight_decay=0.0*)

Implements the Lamb(Layer-wise Adaptive Moments optimizer for Batching training) algorithm.

LAMB is an optimization algorithm employing a layerwise adaptive large batch optimization technique. Refer to the paper LARGE BATCH OPTIMIZATION FOR DEEP LEARNING: TRAINING BERT IN 76 MINUTES.

The LAMB optimizer aims to increase the training batch size without reducing the accuracy, and it supports adaptive element-by-element update and accurate layered correction.

The updating of parameters follows:

Parameters : 1^{st} moment vector m, 2^{nd} moment vector v,

learning rate $\{\gamma_t\}_{t=1}^{T}$, exponential decay rates for the moment estimates β_1 β_2,

scaling function ϕ

Init : $\boldsymbol{m}_0 \leftarrow 0$, $\boldsymbol{v}_0 \leftarrow 0$

---

for t=1 to T **do**

Draw b samples S_t from $\mathbb{P}$.

Compute $g_t = \frac{1}{|\mathbb{S}_t|} \sum_{s_t \in \mathbb{S}_t} \nabla \ell \left(x_t, s_t \right)$.

$\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + \left(1 - \beta_1 \right) \boldsymbol{g}_t$

$\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + \left(1 - \beta_2 \right) \boldsymbol{g}_t^2$

$\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / \left(1 - \beta_1^t \right)$

$\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / \left(1 - \beta_2^t \right)$

Compute ratio $\boldsymbol{r}_t = \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)$

$$\boldsymbol{w}_{t+1}^{(i)} = \boldsymbol{w}_t^{(i)} - \gamma_t \frac{\boldsymbol{\phi}\left(\left\| \boldsymbol{w}_t^{(i)} \right\| \right)}{\left\| \boldsymbol{r}_t^{(i)} + \lambda \boldsymbol{w}_t^{(i)} \right\|} \left(\boldsymbol{r}_t^{(i)} + \lambda \boldsymbol{w}_t^{(i)} \right)$$

end for

---

return $\boldsymbol{w}_{t+1}$

---

m represents the 1st moment vector *moment1*, v represents the 2nd moment vector *moment2*, g represents *gradients*, β_1, β_2 represent *beta1* and *beta2*, t represents the current step while $beta_1^t$ and $beta_2^t$ represent *beta1_power* and *beta2_power*, γ represents *learning_rate*, w represents *params*, ϵ represents *eps*, λ represents *weight_decay*.

Note: There is usually no connection between a optimizer and mixed precision. But when *FixedLossScaleManager* is used and *drop_overflow_update* in *FixedLossScaleManager* is set to False, optimizer needs to set the 'loss_scale'. As this optimizer has no argument of *loss_scale*, so *loss_scale* needs to be processed by other means. Refer document LossScale to process *loss_scale* correctly.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma'

in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Warning: The update process of the Lamb optimizer is not completely elementwise, and the sharding of weights in distributed parallel may affect the update result.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule]*) –
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **beta1** (*float*) –The exponential decay rate for the 1st moment estimations. Default: `0.9`. Should be in range (0.0, 1.0).
- **beta2** (*float*) –The exponential decay rate for the 2nd moment estimations. Default: `0.999`. Should be in range (0.0, 1.0).
- **eps** (*float*) –Term added to the denominator to improve numerical stability. Default: `1e-6`. Should be greater than 0.
- **weight_decay** (*Union[float, int, Cell]*) –Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.

  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

tuple[bool], all elements are `True` .

Raises

- **TypeError** –If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** –If element of *parameters* is neither Parameter nor dict.
- **TypeError** –If *beta1*, *beta2* or *eps* is not a float.
- **TypeError** –If *weight_decay* is neither float nor int.
- **ValueError** –If *eps* is less than or equal to 0.
- **ValueError** –If *beta1*, *beta2* is not in range (0.0, 1.0).
- **ValueError** –If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Lamb(params=net.trainable_params(), learning_rate=0.1)
>>>
>>> #2) Use parameter groups and set different values
>>> poly_decay_lr = nn.PolynomialDecayLR(learning_rate=0.1, end_learning_rate=0.01,
...                                                         decay_steps=4, power = 0.5)
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': poly_decay_lr},
...                 {'order_params': net.trainable_params(0.01)}]
>>> optim = nn.Lamb(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use dynamic learning rate of poly decay learning␣
 ↪rate and default
>>> # weight decay of 0.0 and grad centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
```

(continues on next page)

(continued from previous page)

```
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.11 mindspore.nn.LARS

class mindspore.nn.**LARS**(*optimizer*, *epsilon=1e-05*, *coefficient=0.001*, *use_clip=False*, *lars_filter=lambda x: ...*)

Implements the LARS algorithm.

LARS is an optimization algorithm employing a large batch optimization technique. Refer to paper LARGE BATCH TRAINING OF CONVOLUTIONAL NETWORKS.

The updating formulas are as follows,

Parameters : base learning rate γ_0, momentum m, weight decay λ,
LARS coefficient η, number of steps T
Init : t=0, v=0, init weight w_0^l for each layer l

---

while t<T for each layer l **do**
$g_t^l \leftarrow \nabla L\left(w_t^l\right)$
$\gamma_t \leftarrow \gamma_0 * \left(1-\frac{t}{T}\right)^2$
$\gamma^l \leftarrow \eta * \frac{\|w_t^l\|}{\|g_t^l\|+\lambda\|w_t^l\|}$(compute the local LR $\gamma^{l)}$
$v_{t+1}^l \leftarrow m v_t^l + \gamma_{t+1} * \gamma^l * \left(g_t^l + \lambda w_t^l\right)$
$w_{t+1}^l \leftarrow w_t^l - v_{t+1}^l$
end while

---

w represents the network's params, g represents *gradients*, t represents the current step, λ represents *weight_decay* in *optimizer*, γ represents *learning_rate* in *optimizer*, η represents *coefficient*.

Parameters

- **optimizer** (*mindspore.nn.Optimizer*) –MindSpore optimizer for which to wrap and modify gradients.
- **epsilon** (*float*) –Term added to the denominator to improve numerical stability. Default: `1e-05` .
- **coefficient** (*float*) –Trust coefficient for calculating the local learning rate. Default: `0.001` .
- **use_clip** (*bool*) –Whether to use clip operation for calculating the local learning rate. Default: `False` .
- **lars_filter** (*Function*) –A function to determine which of the network parameters to use LARS algorithm. Default: `lambda x: 'LayerNorm' not in x.name and 'bias' not in x.name`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params* in the optimizer, the shape is the as same as the *params* in the optimizer.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> opt = nn.Momentum(net.trainable_params(), 0.1, 0.9)
>>> opt_lars = nn.LARS(opt, epsilon=1e-08, coefficient=0.02)
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=opt_lars, metrics=None)
```

3.15.12 mindspore.nn.LazyAdam

class mindspore.nn.**LazyAdam**(*params*, *learning_rate=1e-3*, *beta1=0.9*, *beta2=0.999*, *eps=1e-8*, *use_locking=False*, *use_nesterov=False*, *weight_decay=0.0*, *loss_scale=1.0*)

Implements the Adaptive Moment Estimation (Adam) algorithm. The Adam algorithm is proposed in Adam: A Method for Stochastic Optimization.

This optimizer will apply a lazy adam algorithm when gradient is sparse.

The updating formulas are as follows,

$$
\begin{aligned}
m_{t+1} &= \beta_1 * m_t + (1 - \beta_1) * g \\
v_{t+1} &= \beta_2 * v_t + (1 - \beta_2) * g * g \\
\widehat{m_{t+1}} &= \frac{m_{t+1}}{1-\beta_1^t} \\
\widehat{v_{t+1}} &= \frac{v_{t+1}}{1-\beta_2^t} \\
w_{t+1} &= w_t - \gamma * \frac{\widehat{m_{t+1}}}{\sqrt{\widehat{v_{t+1}}}+\epsilon}
\end{aligned}
$$

m represents the 1st moment vector *moment1*, v represents the 2nd moment vector *moment2*, g represents *gradients*, γ represents *learning_rate*, β_1, β_2 represent *beta1* and *beta2*, t represents the current step while $beta_1^t$ and $beta_2^t$ represent *beta1_power* and *beta2_power*, w represents *params*, ϵ represents *eps*.

Note: The sparse strategy is applied while the SparseGatherV2 operator is used for forward network. If the sparse strategy wants to be executed on the host, set the target to the CPU. Please note, the optimizer only updates the current index position of the network parameters when the gradient is sparse. The sparse behavior is not equivalent to the original Adam algorithm. If you want to execute a sparse policy, target needs to be set to CPU.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.

  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.

- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `1e-3`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **beta1** (*float, optional*) – The exponential decay rate for the 1st moment estimations. Should be in range (0.0, 1.0). Default: `0.9`.
- **beta2** (*float, optional*) – The exponential decay rate for the 2nd moment estimations. Should be in range (0.0, 1.0). Default: `0.999`.
- **eps** (*float, optional*) – Term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-8`.
- **use_locking** (*bool, optional*) – Whether to enable a lock to protect the updating process of variable tensors. If `true`, updates of the *w*, *m*, and *v* tensors will be protected by a lock. If `false`, the result is unpredictable. Default: `False`.
- **use_nesterov** (*bool, optional*) – Whether to use Nesterov Accelerated Gradient (NAG) algorithm to update the gradients. If `true`, update the gradients using NAG. If `false`, update the gradients without using NAG. Default: `False`.
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.
- **loss_scale** (*float, optional*) – A floating point value for the loss scale. Should be equal to or greater than 1. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is `True` .

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *beta1*, *beta2*, *eps* or *loss_scale* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **TypeError** – If *use_locking* or *use_nesterov* is not a bool.
- **ValueError** – If *loss_scale* or *eps* is less than or equal to 0.
- **ValueError** – If *beta1*, *beta2* is not in range (0.0, 1.0).
- **ValueError** – If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.LazyAdam(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.LazyAdam(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.13 mindspore.nn.Momentum

class mindspore.nn.**Momentum**(*params*, *learning_rate*, *momentum*, *weight_decay=0.0*, *loss_scale=1.0*, *use_nesterov=False*)

Implements the Momentum algorithm.

Refer to the paper On the importance of initialization and momentum in deep learning for more details.

$$v_{t+1} = v_t * u + grad$$

If use_nesterov is True:

$$p_{t+1} = p_t - (grad * lr + v_{t+1} * u * lr)$$

If use_nesterov is False:

$$p_{t+1} = p_t - lr * v_{t+1}$$

Here: where $grad$, lr, p, v and u denote the gradients, learning_rate, params, moments, and momentum respectively.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) –
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.

  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.

- **momentum** (*float, optional*) – Hyperparameter of type float, means momentum for the moving average. It must be at least 0.0.
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.
- **loss_scale** (*float, optional*) – A floating point value for the loss scale. It must be greater than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **use_nesterov** (*bool, optional*) – Enable Nesterov momentum. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

tuple[bool]. All elements are `True`.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *loss_scale* or *momentum* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **TypeError** – If *use_nesterov* is not a bool.
- **ValueError** – If *loss_scale* is less than or equal to 0.
- **ValueError** – If *weight_decay* or *momentum* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>>
>>> #2) Use parameter groups and set different values
```

(continues on next page)

(continued from previous page)

```
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.Momentum(group_params, learning_rate=0.1, momentum=0.9, weight_decay=0.0)
>>> # The conv_params's parameters will use a learning rate of default value 0.1 and a␣
 ↪weight decay of 0.01 and
>>> # grad centralization of True.
>>> # The no_conv_params's parameters will use a learning rate of 0.01 and a weight decay of␣
 ↪default value 0.0
>>> # and grad centralization of False..
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim, metrics=None)
```

3.15.14 mindspore.nn.OptTFTWrapper

class mindspore.nn.**OptTFTWrapper**(*opt*, ***kwargs*)

Implements TFT optimizer wrapper, this wrapper is used to report status to MindIO TFT before optimizer updating.

Note: This optimizer is depend on MindIO TFT feature. Currently only support ascend graph mode and sink_size must be less than 1.

Parameters

opt (Optimizer) – Must be sub-class of Optimizer.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of opt's *params*, the shape is the same as opt's *params*.

Outputs:

Tensor, result of executing optimizer 'opt'.

Raises

- **TypeError** – If the parameter opt is not an subclass of Optimizer.
- **ValueError** – If the platform is not Ascend graph mode, or customer doesn't switch on TFT feature.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.SGD(params=net.trainable_params())
>>> optim_wrapper = nn.OptTFTWrapper(optim)
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.15 mindspore.nn.ProximalAdagrad

class mindspore.nn.**ProximalAdagrad**(*params*, *accum=0.1*, *learning_rate=0.001*, *l1=0.0*, *l2=0.0*, *use_locking=False*, *loss_scale=1.0*, *weight_decay=0.0*)

Implements the ProximalAdagrad algorithm that is an online Learning and Stochastic Optimization. Refer to paper Efficient Learning using Forward-Backward Splitting.

$$accum_{t+1} = accum_t + g * g$$

$$\text{prox\_v} = w_t - \gamma * g * \frac{1}{\sqrt{accum_{t+1}}}$$

$$w_{t+1} = \frac{sign(\text{prox\_v})}{1 + \gamma * l2} * \max(|\text{prox\_v}| - \gamma * l1, 0)$$

Here : where g , γ, w , $accum$ and t denote the *grads*, *learning_rate*, *params*, accumulation and current step respectively.

Note: The sparse strategy is applied while the SparseGatherV2 operator is used for forward network. If the sparse strategy wants to be executed on the host, set the target to the CPU. The sparse feature is under continuous development.

If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string *"params"*, *"lr"*, *"weight_decay"*, *"grad_centralization"* and *"order_params"* are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.

  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters are grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.

- **accum** (*float, optional*) – The starting value for accumulators *accum*, must be zero or positive values. Default: `0.1`.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `0.001`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of the current step.
- **l1** (*float, optional*) – l1 regularization strength, must be greater than or equal to zero. Default: `0.0`.
- **l2** (*float, optional*) – l2 regularization strength, must be greater than or equal to zero. Default: `0.0`.
- **use_locking** (*bool, optional*) – If `True`, use locks for updating operation. Default: `False`.
- **loss_scale** (*float, optional*) – Value for the loss scale. It must be greater than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False` , then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **weight_decay** (*Union[float, int, Cell], optional*) – Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **grads** (tuple[Tensor]) - The gradients of *params* in the optimizer, the shape is the same as the *params* in optimizer.

Outputs:

Tensor[bool], the value is True.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *accum*, *l1*, *l2* or *loss_scale* is not a float.
- **TypeError** – If *weight_decay* is neither float nor int.
- **ValueError** – If *loss_scale* is less than or equal to 0.

- **ValueError** – If *accum*, *l1*, *l2* or *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.ProximalAdagrad(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.ProximalAdagrad(group_params, learning_rate=0.1, weight_decay=0.0)
 >>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay␣
 ↪of 0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.16 mindspore.nn.RMSProp

class mindspore.nn.**RMSProp**(*params*, *learning_rate=0.1*, *decay=0.9*, *momentum=0.0*, *epsilon=1e-10*, *use_locking=False*, *centered=False*, *loss_scale=1.0*, *weight_decay=0.0*)

Implements Root Mean Squared Propagation (RMSProp) algorithm.

Update *params* according to the RMSProp algorithm. The 29th of the original presentation slide proposes RMSProp. The equation is as follows:

$$s_{t+1} = \rho s_t + (1-\rho)(\nabla Q_i(w))^2$$

$$m_{t+1} = \beta m_t + \frac{\eta}{\sqrt{s_{t+1} + \epsilon}} \nabla Q_i(w)$$

$$w = w - m_{t+1}$$

The first equation calculates moving average of the squared gradient for each weight. Then dividing the gradient by $\sqrt{ms_{t+1} + \epsilon}$.

If centered is True:

$$g_{t+1} = \rho g_t + (1 - \rho)\nabla Q_i(w)$$

$$s_{t+1} = \rho s_t + (1 - \rho)(\nabla Q_i(w))^2$$

$$m_{t+1} = \beta m_t + \frac{\eta}{\sqrt{s_{t+1} - g_{t+1}^2 + \epsilon}}\nabla Q_i(w)$$

$$w = w - m_{t+1}$$

where w represents *params*, which will be updated. g_{t+1} is mean gradients. s_{t+1} is the mean square gradients. m_{t+1} is moment, the delta of w. ρ represents *decay*. β is the momentum term, represents *momentum*. ϵ is a smoothing term to avoid division by zero, represents *epsilon*. η is learning rate, represents *learning_rate*. $\nabla Q_i(w)$ is gradients, represents *gradients*. t represents the current step.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "weight_decay", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters are grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `0.1`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of the current step.

- **decay** (*float, optional*) –Decay rate. Should be equal to or greater than 0. Default: `0.9`.
- **momentum** (*float, optional*) –Hyperparameter of type float, means momentum for the moving average. Should be equal to or greater than 0. Default: `0.0`.
- **epsilon** (*float, optional*) –Term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-10`.
- **use_locking** (*bool, optional*) –Whether to enable a lock to protect the updating process of variable tensors. Default: `False`.
- **centered** (*bool, optional*) –If True, gradients are normalized by the estimated variance of the gradient. Default: `False`.
- **loss_scale** (*float, optional*) –A floating point value for the loss scale. Should be greater than 0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.
- **weight_decay** (*Union[float, int, Cell], optional*) –Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is `True`.

Raises

- **TypeError** –If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** –If *decay*, *momentum*, *epsilon* or *loss_scale* is not a float.
- **TypeError** –If element of *parameters* is neither Parameter nor dict.
- **TypeError** –If *weight_decay* is neither float nor int.
- **TypeError** –If *use_locking* or *centered* is not a bool.
- **ValueError** –If *epsilon* is less than or equal to 0.
- **ValueError** –If *decay* or *momentum* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.RMSProp(params=net.trainable_params(), learning_rate=0.1)
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params, 'weight_decay': 0.01, 'grad_centralization':True}
 ↪,
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.RMSProp(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and weight decay of␣
 ↪0.01 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
 ↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.17 mindspore.nn.Rprop

class mindspore.nn.**Rprop**(*params*, *learning_rate=0.1*, *etas=(0.5, 1.2)*, *step_sizes=(1e-6, 50.)*, *weight_decay=0.*)

Implements Resilient backpropagation.

Further information about this implementation can be found at A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm.

The updating formulas are as follows:

$$
\begin{aligned}
&\textbf{if } g_{t-1} g_t > 0 \\
&\qquad \Delta_t \leftarrow \min(\Delta_{t-1} \eta_+, \Delta_{max}) \\
&\qquad \textbf{else if } g_{t-1} g_t < 0 \\
&\qquad \Delta_t \leftarrow \max(\Delta_{t-1} \eta_-, \Delta_{min}) \\
&\textbf{else} \\
&\qquad \Delta_t \leftarrow \Delta_{t-1} \\
&\qquad w_t \leftarrow w_{t-1} - \Delta_t \operatorname{sign}(g_t)
\end{aligned}
$$

g represents *gradients*, w represents *parameters*, $\Delta_{min/max}$ represents the min/max step size, $\eta_{+/-}$ represents the factors of etaminus and etaplus.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the parameters without 'beta' or 'gamma' in their names.

Users can group parameters to change the strategy of decaying weight.

When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) –Must be list of *Parameter* or list of *dict*. When the *parameters* is a list of *dict*, the *"params"*, *"lr"*, *"weight_decay"*, *"grad_centralization"* and *"order_params"* are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay can be a constant value or a Cell. It is a Cell only when dynamic weight decay is applied. Dynamic weight decay is similar to dynamic learning rate, users need to customize a weight decay schedule only with global step as input, and during training, the optimizer calls the instance of WeightDecaySchedule to get the weight decay value of current step.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) –Learning_rate. Default: `0.1`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.
  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.
- **etas** (*tuple[float, float], optional*) –The factor of multiplicative increasing or descreasing(etaminus, etaplus). Default: `(0.5, 1.2)`.
- **step_sizes** (*tuple[float, float], optional*) –The allowed minimal and maximal step size(min_step_sizes, max_step_size). Default: `(1e-6, 50.)`.
- **weight_decay** (*Union[float, int, Cell], optional*) –Weight decay (L2 penalty). Default: `0.0`.
  - float: The fixed weight decay value. Must be equal to or greater than 0.
  - int: The fixed weight decay value. Must be equal to or greater than 0. It will be converted to float.
  - Cell: Weight decay is dynamic. During training, the optimizer calls the instance of the Cell with step as the input to get the weight decay value of current step.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is True.

Raises

- **TypeError** – If *learning_rate* is not one of int, float, Tensor, Iterable, LearningRateSchedule.
- **TypeError** – If element of *parameters* is neither Parameter nor dict.
- **TypeError** – If *step_sizes* or *etas* is not a tuple.
- **ValueError** – If maximal step size is less than minimal step size.
- **ValueError** – If the length of *step_sizes* or *etas* is not equal to 2.
- **TypeError** – If the element in *etas* or *step_sizes* is not a float.
- **ValueError** – If *etaminus* is not in the range of (0, 1) or *etaplus* is not greater than 1.
- **TypeError** – If *weight_decay* is neither float nor int.
- **ValueError** – If *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.Rprop(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params,'grad_centralization':True},
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.Rprop(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 default weight␣
↪decay of 0.0 and grad
>>> # centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
↪decay of 0.0 and grad
>>> # centralization of False.
>>> # The final parameters order in which the optimizer will be followed is the value of
↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.15.18 mindspore.nn.SGD

class mindspore.nn.**SGD**(*params*, *learning_rate=0.1*, *momentum=0.0*, *dampening=0.0*, *weight_decay=0.0*, *nesterov=False*, *loss_scale=1.0*)

Implements stochastic gradient descent. Momentum is optional.

Introduction to SGD can be found at SGD . Nesterov momentum is based on the formula from paper On the importance of initialization and momentum in deep learning.

$$v_{t+1} = u * v_t + gradient * (1 - dampening)$$

If nesterov is True:

$$p_{t+1} = p_t - lr * (gradient + u * v_{t+1})$$

If nesterov is False:

$$p_{t+1} = p_t - lr * v_{t+1}$$

To be noticed, for the first step, $v_{t+1} = gradient$.

Here : where p, v and u denote the parameters, accum, and momentum respectively.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied.

Parameters

- **params** (*Union[list[Parameter], list[dict]]*) – Must be list of *Parameter* or list of *dict*. When the *params* is a list of *dict*, the string "params", "lr", "grad_centralization" and "order_params" are the keys can be parsed.
  - params: Required. Parameters in current group. The value must be a list of *Parameter*.
  - lr: Optional. If "lr" in the keys, the value of corresponding learning rate will be used. If not, the *learning_rate* in optimizer will be used. Fixed and dynamic learning rate are supported.
  - weight_decay: Optional. If "weight_decay" in the keys, the value of corresponding weight decay will be used. If not, the *weight_decay* in the optimizer will be used. It should be noted that weight decay must be float, dynamic weight decay is currently not supported.
  - grad_centralization: Optional. Must be Boolean. If "grad_centralization" is in the keys, the set value will be used. If not, the *grad_centralization* is False by default. This configuration only works on the convolution layer.
  - order_params: Optional. When parameters is grouped, this usually is used to maintain the order of parameters that appeared in the network to improve performance. The value should be parameters whose order will be followed in optimizer. If *order_params* in the keys, other keys will be ignored and the element of 'order_params' must be in one group of *params*.
- **learning_rate** (*Union[float, int, Tensor, Iterable, LearningRateSchedule], optional*) – Default: `0.1`.
  - float: The fixed learning rate value. Must be equal to or greater than 0.
  - int: The fixed learning rate value. Must be equal to or greater than 0. It will be converted to float.
  - Tensor: Its value should be a scalar or a 1-D vector. For scalar, fixed learning rate will be applied. For vector, learning rate is dynamic, then the i-th step will take the i-th value as the learning rate.
  - Iterable: Learning rate is dynamic. The i-th step will take the i-th value as the learning rate.

  - LearningRateSchedule: Learning rate is dynamic. During training, the optimizer calls the instance of LearningRateSchedule with step as the input to get the learning rate of current step.

- **momentum** (*float, optional*) – A floating point value the momentum. must be at least 0.0. Default: `0.0`.
- **dampening** (*float, optional*) – A floating point value of dampening for momentum. must be at least 0.0. Default: `0.0`.
- **weight_decay** (*float, optional*) – Weight decay (L2 penalty). It must be equal to or greater than 0. Default: `0.0`.
- **nesterov** (*bool, optional*) – Enables the Nesterov momentum. If use nesterov, momentum must be positive, and dampening must be equal to 0.0. Default: `False`.
- **loss_scale** (*float, optional*) – A floating point value for the loss scale, which must be larger than 0.0. In general, use the default value. Only when *FixedLossScaleManager* is used for training and the *drop_overflow_update* in *FixedLossScaleManager* is set to `False`, then this value needs to be the same as the *loss_scale* in *FixedLossScaleManager*. Refer to class *mindspore.amp.FixedLossScaleManager* for more details. Default: `1.0`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*, the shape is the same as *params*.

Outputs:

Tensor[bool], the value is `True`.

Raises

ValueError – If the momentum, dampening or weight_decay value is less than 0.0.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> #1) All parameters use the same learning rate and weight decay
>>> optim = nn.SGD(params=net.trainable_params())
>>>
>>> #2) Use parameter groups and set different values
>>> conv_params = list(filter(lambda x: 'conv' in x.name, net.trainable_params()))
>>> no_conv_params = list(filter(lambda x: 'conv' not in x.name, net.trainable_params()))
>>> group_params = [{'params': conv_params,'grad_centralization':True},
...                 {'params': no_conv_params, 'lr': 0.01},
...                 {'order_params': net.trainable_params()}]
>>> optim = nn.SGD(group_params, learning_rate=0.1, weight_decay=0.0)
>>> # The conv_params's parameters will use default learning rate of 0.1 and default weight␣
↪decay of 0.0
>>> # and grad centralization of True.
>>> # The no_conv_params's parameters will use learning rate of 0.01 and default weight␣
↪decay of 0.0 and grad
>>> # centralization of False.
```

(continues on next page)

(continued from previous page)

```
>>> # The final parameters order in which the optimizer will be followed is the value of
 ↪'order_params'.
>>>
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim)
```

3.16 Dynamic Learning Rate

3.16.1 LearningRateSchedule Class

The dynamic learning rates in this module are all subclasses of LearningRateSchedule. Pass the instance of LearningRateSchedule to an optimizer. During the training process, the optimizer calls the instance taking current step as input to get the current learning rate.

```
import mindspore.nn as nn

min_lr = 0.01
max_lr = 0.1
decay_steps = 4
cosine_decay_lr = nn.CosineDecayLR(min_lr, max_lr, decay_steps)

net = Net()
optim = nn.Momentum(net.trainable_params(), learning_rate=cosine_decay_lr, momentum=0.9)
```

API Name	Description	Supported Platforms
mindspore.nn.CosineDecayLR	Calculates learning rate based on cosine decay function.	Ascend GPU
mindspore.nn.ExponentialDecayLR	Calculates learning rate based on exponential decay function.	Ascend GPU CPU
mindspore.nn.InverseDecayLR	Calculates learning rate base on inverse-time decay function.	Ascend GPU CPU
mindspore.nn.NaturalExpDecayLR	Calculates learning rate base on natural exponential decay function.	Ascend GPU CPU
mindspore.nn.PolynomialDecayLR	Calculates learning rate base on polynomial decay function.	Ascend GPU
mindspore.nn.WarmUpLR	Gets learning rate warming up.	Ascend GPU CPU

mindspore.nn.CosineDecayLR

class mindspore.nn.**CosineDecayLR**(*min_lr*, *max_lr*, *decay_steps*)

Calculates learning rate based on cosine decay function.

For current step, the formula of computing decayed learning rate is:

$$decayed\_learning\_rate = min\_lr + 0.5 * (max\_lr - min\_lr) * (1 + cos(\frac{current\_step}{decay\_steps}\pi))$$

Parameters

- **min_lr** (*float*) – The minimum value of learning rate.
- **max_lr** (*float*) – The maximum value of learning rate.

- **decay_steps** (*int*) –Number of steps to decay over.

Inputs:

- **global_step** (Tensor) - The current step number.

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *min_lr* or *max_lr* is not a float.
- **TypeError** –If *decay_steps* is not an int.
- **ValueError** –If *min_lr* is less than 0 or *decay_steps* is less than 1.
- **ValueError** –If *max_lr* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> min_lr = 0.01
>>> max_lr = 0.1
>>> decay_steps = 4
>>> global_steps = Tensor(2, mindspore.int32)
>>> cosine_decay_lr = nn.CosineDecayLR(min_lr, max_lr, decay_steps)
>>> lr = cosine_decay_lr(global_steps)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.ExponentialDecayLR

class mindspore.nn.**ExponentialDecayLR**(*learning_rate*, *decay_rate*, *decay_steps*, *is_stair=False*)

Calculates learning rate based on exponential decay function.

For current step, the formula of computing decayed learning rate is:

$$decayed\_learning\_rate = learning\_rate * decay\_rate^{p}$$

Where :

$$p = \frac{current\_step}{decay\_steps}$$

If *is_stair* is True, the formula is :

$$p = floor(\frac{current\_step}{decay\_steps})$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.

- **decay_rate** (*float*) –The decay rate.
- **decay_steps** (*int*) –Number of steps to decay over.
- **is_stair** (*bool, optional*) –If `True`, learning rate is decayed once every *decay_steps* time. Default: `False` .

Inputs:

- **global_step** (Tensor) - The current step number. $current\_step$ in the above formula.

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **TypeError** –If *decay_steps* is not an int or *is_stair* is not a bool.
- **ValueError** –If *decay_steps* is less than 1.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.9
>>> decay_steps = 4
>>> global_step = Tensor(2, mindspore.int32)
>>> exponential_decay_lr = nn.ExponentialDecayLR(learning_rate, decay_rate, decay_steps)
>>> lr = exponential_decay_lr(global_step)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.InverseDecayLR

class mindspore.nn.**InverseDecayLR**(*learning_rate*, *decay_rate*, *decay_steps*, *is_stair=False*)

Calculates learning rate base on inverse-time decay function.

For current step, the formula of computing decayed learning rate is:

$$decayed\_learning\_rate = learning\_rate/(1 + decay\_rate * p)$$

Where :

$$p = \frac{current\_step}{decay\_steps}$$

If *is_stair* is True, the formula is :

$$p = floor(\frac{current\_step}{decay\_steps})$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **decay_rate** (*float*) –The decay rate.
- **decay_steps** (*int*) –Number of steps to decay over.
- **is_stair** (*bool, optional*) –If true, learning rate decay once every *decay_steps* times. If False, the learning rate decays for every step. Default: `False` .

Inputs:

- **global_step** (Tensor) - The current step number.

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **TypeError** –If *decay_steps* is not an int or *is_stair* is not a bool.
- **ValueError** –If *decay_steps* is less than 1.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.9
>>> decay_steps = 4
>>> global_step = Tensor(2, mindspore.int32)
>>> inverse_decay_lr = nn.InverseDecayLR(learning_rate, decay_rate, decay_steps, True)
>>> lr = inverse_decay_lr(global_step)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.NaturalExpDecayLR

class mindspore.nn.**NaturalExpDecayLR**(*learning_rate*, *decay_rate*, *decay_steps*, *is_stair=False*)

Calculates learning rate base on natural exponential decay function.

For current step, the formula of computing decayed learning rate is:

$$decayed\_learning\_rate = learning\_rate * e^{-decay\_rate*p}$$

Where :

$$p = \frac{current\_step}{decay\_steps}$$

If *is_stair* is True, the formula is :

$$p = floor(\frac{current\_step}{decay\_steps})$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **decay_rate** (*float*) –The decay rate.
- **decay_steps** (*int*) –Number of steps to decay over.
- **is_stair** (*bool*) –If `true` , learning rate is decayed once every *decay_steps* time. Default: `False` .

Inputs:

- **global_step** (Tensor) - The current step number.

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **TypeError** –If *decay_steps* is not an int or *is_stair* is not a bool.
- **ValueError** –If *decay_steps* is less than 1.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.9
>>> decay_steps = 4
>>> global_step = Tensor(2, mindspore.int32)
>>> natural_exp_decay_lr = nn.NaturalExpDecayLR(learning_rate, decay_rate, decay_steps, True)
>>> lr = natural_exp_decay_lr(global_step)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.PolynomialDecayLR

class mindspore.nn.**PolynomialDecayLR**(*learning_rate*, *end_learning_rate*, *decay_steps*, *power*, *update_decay_steps=False*)

Calculates learning rate base on polynomial decay function.

For current step, the formula of computing decayed learning rate is:

$$\begin{aligned} decayed\_learning\_rate =& (learning\_rate - end\_learning\_rate) * \\ & (1 - tmp\_step / tmp\_decay\_steps)^{power} \\ & + end\_learning\_rate \end{aligned}$$

Where :

$$tmp\_step = \min(current\_step, decay\_steps)$$

If *update_decay_steps* is true, update the value of tmp_decay_steps every *decay_steps*. The formula is :

$$tmp\_decay\_steps = decay\_steps * ceil(current\_step / decay\_steps)$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **end_learning_rate** (*float*) –The end value of learning rate.
- **decay_steps** (*int*) –Number of steps to decay over.
- **power** (*float*) –The power of polynomial. It must be greater than 0.
- **update_decay_steps** (*bool*) –If `true` , learning rate is decayed once every *decay_steps* time. Default: `False` .

Inputs:

- **global_step** (Tensor) - The current step number. Shape is ().

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *learning_rate*, *end_learning_rate* or *power* is not a float.
- **TypeError** –If *decay_steps* is not an int or *update_decay_steps* is not a bool.
- **ValueError** –If *end_learning_rate* is less than 0 or *decay_steps* is less than 1.
- **ValueError** –If *learning_rate* or *power* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> learning_rate = 0.1
>>> end_learning_rate = 0.01
>>> decay_steps = 4
>>> power = 0.5
>>> global_step = Tensor(2, mindspore.int32)
>>> polynomial_decay_lr = nn.PolynomialDecayLR(learning_rate, end_learning_rate, decay_steps,
 ↪ power)
>>> lr = polynomial_decay_lr(global_step)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.WarmUpLR

class mindspore.nn.**WarmUpLR**(*learning_rate*, *warmup_steps*)

Gets learning rate warming up.

For current step, the formula of computing warmup learning rate is:

$$warmup\_learning\_rate = learning\_rate * tmp\_step / warmup\_steps$$

Where

$$tmp\_step = \min(current\_step, warmup\_steps)$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate. The value of *learning_rate* must be greater than 0.
- **warmup_steps** (*int*) –The warm up steps of learning rate. The value of *warmup_steps* must be greater than or equal to 1.

Inputs:

- **global_step** (Tensor) - The current step number. Shape is ().

Outputs:

Tensor. The learning rate value for the current step with shape ().

Raises

- **TypeError** –If *learning_rate* is not a float.
- **TypeError** –If *warmup_steps* is not an int.
- **ValueError** –If *warmup_steps* is less than 1.
- **ValueError** –If *learning_rate* is less than or equal to 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>>
>>> learning_rate = 0.1
>>> warmup_steps = 2
>>> global_step = Tensor(2, mindspore.int32)
>>> warmup_lr = nn.WarmUpLR(learning_rate, warmup_steps)
>>> lr = warmup_lr(global_step)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

3.16.2 Dynamic LR Function

The dynamic learning rates in this module are all functions. Call the function and pass the result to an optimizer. During the training process, the optimizer takes result[current step] as current learning rate.

```
import mindspore.nn as nn

min_lr = 0.01
max_lr = 0.1
total_step = 6
step_per_epoch = 1
decay_epoch = 4

lr= nn.cosine_decay_lr(min_lr, max_lr, total_step, step_per_epoch, decay_epoch)

net = Net()
optim = nn.Momentum(net.trainable_params(), learning_rate=lr, momentum=0.9)
```

API Name	Description	Supported Platforms
mindspore.nn.cosine_decay_lr	Calculates learning rate base on cosine decay function.	Ascend GPU CPU
mindspore.nn.exponential_decay_lr	Calculates learning rate base on exponential decay function.	Ascend GPU CPU
mindspore.nn.inverse_decay_lr	Calculates learning rate base on inverse-time decay function.	Ascend GPU CPU
mindspore.nn.natural_exp_decay_lr	Calculates learning rate base on natural exponential decay function.	Ascend GPU CPU
mindspore.nn.piecewise_constant_lr	Get piecewise constant learning rate.	Ascend GPU CPU
mindspore.nn.polynomial_decay_lr	Calculates learning rate base on polynomial decay function.	Ascend GPU CPU
mindspore.nn.warmup_lr	Gets learning rate warming up.	Ascend GPU CPU

mindspore.nn.cosine_decay_lr

mindspore.nn.**cosine_decay_lr**(*min_lr*, *max_lr*, *total_step*, *step_per_epoch*, *decay_epoch*)

Calculates learning rate base on cosine decay function. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing decayed_learning_rate[i] is:

$$decayed\_learning\_rate[i] = min\_lr + 0.5 * (max\_lr - min\_lr) * (1 + cos(\frac{current\_epoch}{decay\_epoch}\pi))$$

Where $current\_epoch = floor(\frac{i}{step\_per\_epoch})$.

Parameters

- **min_lr** (*float*) –The minimum value of learning rate.
- **max_lr** (*float*) –The maximum value of learning rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **decay_epoch** (*int*) –Number of epochs to decay over.

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *min_lr* or *max_lr* is not a float.
- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **ValueError** –If *max_lr* is not greater than 0 or *min_lr* is less than 0.
- **ValueError** –If *total_step* or *step_per_epoch* or *decay_epoch* is less than 0.
- **ValueError** –If *min_lr* is greater than or equal to *max_lr*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.nn as nn
>>>
>>> min_lr = 0.01
>>> max_lr = 0.1
>>> total_step = 6
>>> step_per_epoch = 2
>>> decay_epoch = 2
>>> lr = nn.cosine_decay_lr(min_lr, max_lr, total_step, step_per_epoch, decay_epoch)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.exponential_decay_lr

mindspore.nn.**exponential_decay_lr**(*learning_rate*, *decay_rate*, *total_step*, *step_per_epoch*, *decay_epoch*, *is_stair=False*)

Calculates learning rate base on exponential decay function. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing decayed_learning_rate[i] is:

$$decayed\_learning\_rate[i] = learning\_rate * decay\_rate^{\frac{current\_epoch}{decay\_epoch}}$$

Where $current\_epoch = floor(\frac{i}{step\_per\_epoch})$.

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **decay_rate** (*float*) –The decay rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **decay_epoch** (*int*) –Number of epochs to decay over.
- **is_stair** (*bool*) –If true, learning rate is decayed once every *decay_epoch* times. Default: `False`.

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **TypeError** –If *is_stair* is not a bool.
- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.9
>>> total_step = 6
>>> step_per_epoch = 2
>>> decay_epoch = 1
>>> lr = nn.exponential_decay_lr(learning_rate, decay_rate, total_step, step_per_epoch,␣
↪decay_epoch)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.inverse_decay_lr

mindspore.nn.**inverse_decay_lr**(*learning_rate*, *decay_rate*, *total_step*, *step_per_epoch*, *decay_epoch*, *is_stair=False*)

Calculates learning rate base on inverse-time decay function. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing decayed_learning_rate[i] is:

$$decayed\_learning\_rate[i] = learning\_rate/(1 + decay\_rate * current\_epoch/decay\_epoch)$$

Where $current\_epoch = floor(\frac{i}{step\_per\_epoch})$.

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **decay_rate** (*float*) –The decay rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **decay_epoch** (*int*) –Number of epochs to decay over.
- **is_stair** (*bool*) –If true, learning rate is decayed once every *decay_epoch* times. If False, the learning rate decays for every epoch. Default: `False` .

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **TypeError** –If *is_stair* is not a bool.
- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.5
>>> total_step = 6
>>> step_per_epoch = 1
>>> decay_epoch = 1
>>> lr = nn.inverse_decay_lr(learning_rate, decay_rate, total_step, step_per_epoch, decay_
↪epoch, True)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.natural_exp_decay_lr

mindspore.nn.**natural_exp_decay_lr**(*learning_rate*, *decay_rate*, *total_step*, *step_per_epoch*, *decay_epoch*, *is_stair=False*)

Calculates learning rate base on natural exponential decay function. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing decayed_learning_rate[i] is:

$$decayed\_learning\_rate[i] = learning\_rate * e^{-decay\_rate*current\_epoch}$$

Where $current\_epoch = floor(\frac{i}{step\_per\_epoch})$.

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **decay_rate** (*float*) –The decay rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **decay_epoch** (*int*) –Number of epochs to decay over.
- **is_stair** (*bool*) –If true, learning rate is decayed once every *decay_epoch* times. Default: `False` .

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **TypeError** –If *is_stair* is not a bool.
- **TypeError** –If *learning_rate* or *decay_rate* is not a float.
- **ValueError** –If *learning_rate* or *decay_rate* is less than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>>
>>> learning_rate = 0.1
>>> decay_rate = 0.9
>>> total_step = 6
>>> step_per_epoch = 2
>>> decay_epoch = 2
>>> lr = nn.natural_exp_decay_lr(learning_rate, decay_rate, total_step, step_per_epoch,␣
↪decay_epoch, True)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.piecewise_constant_lr

mindspore.nn.**piecewise_constant_lr**(*milestone*, *learning_rates*)

Get piecewise constant learning rate. The learning rate for each step will be stored in a list.

Calculate learning rate by the given *milestone* and *learning_rates*. Let the value of *milestone* be $(M_1, M_2, ..., M_t, ..., M_N)$ and the value of *learning_rates* be $(x_1, x_2, ..., x_t, ..., x_N)$. N is the length of *milestone*. Let the output learning rate be $y[i]$, then for the i-th step, the formula of computing decayed_learning_rate[i] is:

$$y[i] = x_t, \ for \ i \in [M_{t-1}, M_t)$$

Parameters

- **milestone** (*Union[list[int], tuple[int]]*) – A list of milestone. When the specified step is reached, use the corresponding *learning_rates*. This list is a monotone increasing list. Every element in the list must be greater than 0.
- **learning_rates** (*Union[list[float], tuple[float]]*) – A list of learning rates.

Returns

list[float]. The size of list is M_N.

Raises

- **TypeError** – If *milestone* or *learning_rates* is neither a tuple nor a list.
- **ValueError** – If the length of *milestone* and *learning_rates* is not same.
- **ValueError** – If the value in *milestone* is not monotonically decreasing.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.nn as nn
>>>
>>> milestone = [2, 5, 10]
>>> learning_rates = [0.1, 0.05, 0.01]
>>> lr = nn.piecewise_constant_lr(milestone, learning_rates)
>>> # learning_rates = 0.1  if step <= 2
>>> # learning_rates = 0.05  if 2 < step <= 5
>>> # learning_rates = 0.01  if 5 < step <= 10
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.polynomial_decay_lr

mindspore.nn.**polynomial_decay_lr**(*learning_rate*, *end_learning_rate*, *total_step*, *step_per_epoch*, *decay_epoch*, *power*, *update_decay_epoch=False*)

Calculates learning rate base on polynomial decay function. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing decayed_learning_rate[i] is:

$$decayed\_learning\_rate[i] = (learning\_rate - end\_learning\_rate) * (1 - tmp\_epoch/tmp\_decay\_epoch)^{power} + end\_learning\_$$

Where:

$$tmp\_epoch = \min(current\_epoch, decay\_epoch)$$

$$current\_epoch = floor(\frac{i}{step\_per\_epoch})$$

$$tmp\_decay\_epoch = decay\_epoch$$

If *update_decay_epoch* is true, update the value of tmp_decay_epoch every epoch. The formula is:

$$tmp\_decay\_epoch = decay\_epoch * ceil(current\_epoch/decay\_epoch)$$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **end_learning_rate** (*float*) –The end value of learning rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **decay_epoch** (*int*) –Number of epochs to decay over.
- **power** (*float*) –The power of polynomial. It must be greater than 0.
- **update_decay_epoch** (*bool*) –If `true`, update *decay_epoch*. Default: `False` .

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *learning_rate* or *end_learning_rate* or *power* is not a float.
- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **TypeError** –If *update_decay_epoch* is not a bool.
- **ValueError** –If *learning_rate* or *power* is not greater than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.nn as nn
>>>
>>> lr = 0.1
>>> end_learning_rate = 0.01
>>> total_step = 6
>>> step_per_epoch = 2
>>> decay_epoch = 2
>>> power = 0.5
>>> lr = nn.polynomial_decay_lr(lr, end_learning_rate, total_step, step_per_epoch, decay_
↪epoch, power)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

mindspore.nn.warmup_lr

mindspore.nn.**warmup_lr**(*learning_rate*, *total_step*, *step_per_epoch*, *warmup_epoch*)

Gets learning rate warming up. The learning rate for each step will be stored in a list.

For the i-th step, the formula of computing warmup_learning_rate[i] is:

$$warmup\_learning\_rate[i] = learning\_rate * tmp\_epoch / warmup\_epoch$$

Where $tmp\_epoch = \min(current\_epoch, warmup\_epoch)$, $current\_epoch = floor(\frac{i}{step\_per\_epoch})$

Parameters

- **learning_rate** (*float*) –The initial value of learning rate.
- **total_step** (*int*) –The total number of steps.
- **step_per_epoch** (*int*) –The number of steps in per epoch.
- **warmup_epoch** (*int*) –A value that determines the epochs of the learning rate is warmed up.

Returns

list[float]. The size of list is *total_step*.

Raises

- **TypeError** –If *learning_rate* is not a float.
- **TypeError** –If *total_step* or *step_per_epoch* or *decay_epoch* is not an int.
- **ValueError** –If *learning_rate* is less than 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.nn as nn
>>>
>>> learning_rate = 0.1
>>> total_step = 6
>>> step_per_epoch = 2
>>> warmup_epoch = 2
>>> lr = nn.warmup_lr(learning_rate, total_step, step_per_epoch, warmup_epoch)
>>> net = nn.Dense(2, 3)
>>> optim = nn.SGD(net.trainable_params(), learning_rate=lr)
```

3.17 Image Processing Layer

API Name	Description	Supported Platforms
mindspore.nn.PixelShuffle	Applies the PixelShuffle operation over input which implements sub-pixel convolutions with stride $1/r$.	Ascend GPU CPU
mindspore.nn.PixelUnshuffle	Applies the PixelUnshuffle operation over input which is the inverse of PixelShuffle.	Ascend GPU CPU

continues on next page

Table 18 – continued from previous page

mindspore.nn.Upsample	For details, please refer to *mindspore.ops.interpolate()*.	Ascend GPU CPU

3.17.1 mindspore.nn.PixelShuffle

class mindspore.nn.**PixelShuffle**(*upscale_factor*)

Applies the PixelShuffle operation over input which implements sub-pixel convolutions with stride $1/r$. For more details, refer to Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network .

Typically, the input is of shape $(*, C \times r^2, H, W)$, and the output is of shape $(*, C, H \times r, W \times r)$, where r is an upscale factor and $*$ is zero or more batch dimensions.

Note: The dimension of input Tensor on Ascend should be less than 7.

Parameters

upscale_factor (*int*) –factor to shuffle the input, and is a positive integer. *upscale_factor* is the above-mentioned r.

Inputs:

- **input** (Tensor) - Tensor of shape $(*, C \times r^2, H, W)$. The dimension of x is larger than 2, and the length of third to last dimension can be divisible by *upscale_factor* squared.

Outputs:

- **output** (Tensor) - Tensor of shape $(*, C, H \times r, W \times r)$.

Raises

- **ValueError** –If *upscale_factor* is not a positive integer.
- **ValueError** –If the length of third to last dimension of *input* is not divisible by *upscale_factor* squared.
- **ValueError** –If the dimension of *input* is less than 3.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> input_x = np.arange(3 * 2 * 8 * 4 * 4).reshape((3, 2, 8, 4, 4))
>>> input_x = ms.Tensor(input_x, ms.dtype.int32)
>>> pixel_shuffle = ms.nn.PixelShuffle(2)
>>> output = pixel_shuffle(input_x)
>>> print(output.shape)
(3, 2, 2, 8, 8)
```

3.17.2 mindspore.nn.PixelUnshuffle

class mindspore.nn.**PixelUnshuffle**(*downscale_factor*)

Applies the PixelUnshuffle operation over input which is the inverse of PixelShuffle. For more details, refer to Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network .

Typically, the input is of shape $(*, C, H \times r, W \times r)$, and the output is of shape $(*, C \times r^2, H, W)$, where r is a downscale factor and $*$ is zero or more batch dimensions.

Parameters

downscale_factor (*int*) –factor to unshuffle the input, and is a positive integer. *downscale_factor* is the above-mentioned r.

Inputs:

- **input** (Tensor) - Tensor of shape $(*, C, H \times r, W \times r)$. The dimension of *input* is larger than 2, and the length of second to last dimension or last dimension can be divisible by *downscale_factor* .

Outputs:

- **output** (Tensor) - Tensor of shape $(*, C \times r^2, H, W)$.

Raises

- **ValueError** –If *downscale_factor* is not a positive integer.
- **ValueError** –If the length of second to last dimension or last dimension is not divisible by *downscale_factor* .
- **ValueError** –If the dimension of *input* is less than 3.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> pixel_unshuffle = ms.nn.PixelUnshuffle(2)
>>> input_x = np.arange(8 * 8).reshape((1, 1, 8, 8))
>>> input_x = ms.Tensor(input_x, ms.dtype.int32)
>>> output = pixel_unshuffle(input_x)
>>> print(output.shape)
(1, 4, 4, 4)
```

3.17.3 mindspore.nn.Upsample

class mindspore.nn.**Upsample**(*size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*)

For details, please refer to *mindspore.ops.interpolate()*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> x = ms.Tensor([[[[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]]]])
>>> upsample = ms.nn.Upsample(size=(5, 5))
>>> out = upsample(x)
>>> print(x.asnumpy())
[[[[1. 2. 3. 4.]
   [5. 6. 7. 8.]]]]
>>> print(out.asnumpy())
[[[[1. 1. 2. 3. 4.]
   [1. 1. 2. 3. 4.]
   [1. 1. 2. 3. 4.]
   [5. 5. 6. 7. 8.]
   [5. 5. 6. 7. 8.]]]]
>>> print(out.shape)
(1, 1, 5, 5)
```

3.18 Common Layer

API Name	Description	Supported Platforms
mindspore.nn.ChannelShuffle	Divide the channels in a tensor of shape $(*, C, H, W)$ into g group and rearrange them as $(*, \frac{C}{g}, g, H * W)$, while retaining the original tensor shape in the final output.	`Ascend` `GPU` `CPU`
mindspore.nn.Flatten	Flatten the input Tensor along dimensions from *start_dim* to *end_dim*.	`Ascend` `GPU` `CPU`
mindspore.nn.Identity	A placeholder identity operator that returns the same as input.	`Ascend` `GPU` `CPU`
mindspore.nn.Unflatten	Unflattens a Tensor dim according to *axis* and *unflattened_size*.	`Ascend` `GPU` `CPU`

3.18.1 mindspore.nn.ChannelShuffle

class mindspore.nn.**ChannelShuffle**(*groups*)

Divide the channels in a tensor of shape $(*, C, H, W)$ into g group and rearrange them as $(*, \frac{C}{g}, g, H * W)$, while retaining the original tensor shape in the final output.

Parameters

groups (*int*) –Number of groups to divide channels in, must be greater than 0. Refer to g in the above formula.

Inputs:

- **x** (Tensor) - Tensor of shape $(*, C_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

- **TypeError** –If *groups* is not a positive integer.
- **ValueError** –If *groups* is less than 1.
- **ValueError** –If dims of *x* is less than 3.
- **ValueError** –If number of channels can not be divisible by groups.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> channel_shuffle = ms.nn.ChannelShuffle(2)
>>> x = ms.Tensor(np.arange(16).astype(np.int32).reshape(1, 4, 2, 2))
>>> print(x)
[[[[ 0  1]
   [ 2  3]]
  [[ 4  5]
   [ 6  7]]
  [[ 8  9]
   [10 11]]
  [[12 13]
   [14 15]]]]
>>> output = channel_shuffle(x)
>>> print(output)
[[[[ 0  1]
   [ 2  3]]
  [[ 8  9]
   [10 11]]
  [[ 4  5]
   [ 6  7]]
  [[12 13]
   [14 15]]]]
```

3.18.2 mindspore.nn.Flatten

class mindspore.nn.**Flatten**(*start_dim=1*, *end_dim=-1*)

Flatten the input Tensor along dimensions from *start_dim* to *end_dim*.

Parameters

- **start_dim** (*int, optional*) – The first dimension to flatten. Default: `1`.
- **end_dim** (*int, optional*) – The last dimension to flatten. Default: `-1`.

Inputs:

- **x** (Tensor) - The input Tensor to be flattened.

Outputs:

Tensor. If no dimensions are flattened, returns the original *x*, otherwise return the flattened Tensor. If *x* is a 0-dimensional Tensor, a 1-dimensional Tensor will be returned.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *start_dim* or *end_dim* is not int.
- **ValueError** – If *start_dim* is greater than *end_dim* after canonicalized.
- **ValueError** – If *start_dim* or *end_dim* is not in range of [-x.dim, x.dim-1]. For example, the default values are used for the args and the input is a 0-dimensional or 1-dimensional Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> x = Tensor(np.array([[[1.2, 1.2], [2.1, 2.1]], [[2.2, 2.2], [3.2, 3.2]]]), mindspore.
↪float32)
>>> net = nn.Flatten()
>>> output = net(x)
>>> print(output)
[[1.2 1.2 2.1 2.1]
 [2.2 2.2 3.2 3.2]]
>>> print(f"before flatten the x shape is {x.shape}")
before flatten the x shape is  (2, 2, 2)
>>> print(f"after flatten the output shape is {output.shape}")
after flatten the output shape is (2, 4)
```

3.18.3 mindspore.nn.Identity

class mindspore.nn.**Identity**(**args*, ***kwargs*)

A placeholder identity operator that returns the same as input.

Parameters

- **args** (*Any*) – Any argument.
- **kwargs** (*Any*) – Any keyword argument.

Inputs:

- **input** (Any) - The input of Identity.

Outputs:

The same as *input*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.array([1, 2, 3, 4]), mindspore.int64)
>>> net = nn.Identity()
>>> output = net(input)
>>> print(output)
[1 2 3 4]
```

3.18.4 mindspore.nn.Unflatten

class mindspore.nn.**Unflatten**(*axis*, *unflattened_size*)

Unflattens a Tensor dim according to *axis* and *unflattened_size*.

Parameters

- **axis** (*int*) – specifies the dimension of the input Tensor to be unflattened.
- **unflattened_size** (*Union(tuple[int], list[int])*) – the new shape of the unflattened dimension of the Tensor and it can be a tuple of ints or a list of ints. The product of *unflattened_size* must equal to input_shape[axis].

Inputs:

- **input** (Tensor) - The input Tensor to be unflattened.

Outputs:

Tensor that has been unflattend.

Raises

- **TypeError** – If *axis* is not int.
- **TypeError** – If *unflattened_size* is neither tuple of ints nor list of ints.

- **TypeError** – The product of *unflattened_size* does not equal to input_shape[axis].

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, nn
>>> import numpy as np
>>> input = Tensor(np.arange(0, 100).reshape(2, 10, 5), mindspore.float32)
>>> net = nn.Unflatten(1, (2, 5))
>>> output = net(input)
>>> print(f"before unflatten the input shape is {input.shape}")
before unflatten the input shape is  (2, 10, 5)
>>> print(f"after unflatten the output shape is {output.shape}")
after unflatten the output shape is (2, 2, 5, 5)
```

3.19 Tools

mindspore.nn.utils.no_init_parameters	This interface is used to skip parameter initialization.

3.19.1 mindspore.nn.utils.no_init_parameters

mindspore.nn.utils.**no_init_parameters**()

This interface is used to skip parameter initialization.

In scenarios where a checkpoint is loaded, parameters within the network instantiation will be instantiated and occupy physical memory. Loading a checkpoint will replace the parameter values. Decorator can be applied during network instantiation to add an attribute *init_param* to all parameters within the current Cell, setting it to *init_param=False* . When *init_param=False* is detected, the initialization of the parameters is skipped, and the parameters are assigned values directly from the checkpoint during loading, which can optimize performance and reduce physical memory usage.

Note: Initialization of parameters created with *initializer* can only be skipped. Parameters created by *Tensor* or *numpy* cannot be skipped.

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn, ops, load_checkpoint
>>> from mindspore.common.initializer import initializer
>>> from mindspore.nn.utils import no_init_parameters
>>> # 1. Add a decorator to the network that requires delayed initialization
>>> class Net(nn.Cell):
...     def __init__(self, in_channels, out_channels):
...         super().__init__()
...         self.weight = ms.Parameter(initializer("normal", [in_channels, out_channels], ms.
↪float32))
```

(continues on next page)

(continued from previous page)

```
...         self.bias = ms.Parameter(initializer("normal", [out_channels], ms.float32))
...         self.matmul = ops.MatMul()
...         self.add = ops.Add()
...
...     def construct(self, x):
...         x = self.matmul(x, self.weight)
...         x = self.add(x, self.bias)
...         return x
>>> with no_init_parameters():
...     # After instantiation, all parameters in the net are not initialized
...     net = Net(28*28, 64)
>>> # 2. Load checkpoint parameters to the net
>>> load_checkpoint('./checkpoint/test_net.ckpt', net=net)
>>> # 3. After loading the checkpoint, manually call init_parameters_data() to initialize
>>> #    the uninitialized parameters in the net if need. If the network is executed,
>>> #    the framework will automatically call this interface.
>>> net.init_parameters_data()
```

CHAPTER

FOUR

MINDSPORE.MINT

mindspore.mint provides a large number of functional, nn, optimizer interfaces. The API usages and functions are consistent with the mainstream usage in the industry for easy reference. The mint interface is currently an experimental interface and performs better than ops in graph mode of O0 and PyNative mode. Currently, the O2 (graph sinking mode) and CPU/GPU backend are not supported, and it will be gradually improved in the future.

The module import method is as follows:

```
from mindspore import mint
```

Compared with the previous version, the added, deleted and supported platforms change information of *mindspore.mint* operators in MindSpore, please refer to the link mindspore.mint API Interface Change .

4.1 Tensor

4.1.1 Creation Operations

API Name	Description	Supported Platforms
mindspore.mint.arange	Creates a sequence of numbers that begins at *start* and extends by increments of *step* up to but not including *end*.	Ascend
mindspore.mint.bernoulli	Sample from the Bernoulli distribution and randomly set the i^{th} element of the *output* to (0 or 1) according to the i^{th} probability value given in the *input*.	Ascend
mindspore.mint.bincount	Count the occurrences of each value in the input.	Ascend
mindspore.mint.clone	Returns a copy of the input tensor.	Ascend
mindspore.mint.eye	Returns a tensor with ones on the diagonal and zeros in the rest.	Ascend
mindspore.mint.einsum	According to the Einstein summation Convention (Einsum), the product of the input tensor elements is summed along the specified dimension.	Ascend
mindspore.mint.empty	Creates a tensor with uninitialized data, whose shape, dtype and device are described by the argument *size*, *dtype* and *device* respectively.	Ascend
mindspore.mint.empty_like	Returns an uninitialized Tensor with the same shape as the *input*.	Ascend
mindspore.mint.full	Create a Tensor of the specified shape and fill it with the specified value.	Ascend

continues on next page

Table 1 – continued from previous page

mindspore.mint.full_like	Return a Tensor of the same shape as *input* and filled with *fill_value*.	Ascend
mindspore.mint.linspace	Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].	Ascend
mindspore.mint.ones	Creates a tensor filled with value ones.	Ascend
mindspore.mint.ones_like	Creates a tensor filled with 1, with the same shape as input, and its data type is determined by the given dtype.	Ascend
mindspore.mint.randint	Returns a new tensor filled with integer numbers from the uniform distribution over an interval $[low, high)$ based on the given shape and dtype.	Ascend
mindspore.mint.randint_like	Returns a new tensor filled with integer numbers from the uniform distribution over an interval $[low, high)$ based on the given dtype and shape of the input tensor.	Ascend
mindspore.mint.randn	Returns a new tensor filled with numbers from the normal distribution over an interval $[0, 1)$ based on the given shape and dtype.	Ascend
mindspore.mint.randn_like	Returns a new tensor filled with numbers from the normal distribution over an interval $[0, 1)$ based on the given dtype and shape of the input tensor.	Ascend
mindspore.mint.randperm	Generates random permutation of integers from 0 to n-1.	Ascend
mindspore.mint.zeros	Creates a tensor filled with 0 with shape described by *size* and fills it with value 0 in type of *dtype*.	Ascend
mindspore.mint.zeros_like	Creates a tensor filled with 0, with the same size as input.	Ascend

mindspore.mint.arange

mindspore.mint.**arange**(*start=0*, *end=None*, *step=1*, *, *dtype=None*)

Creates a sequence of numbers that begins at *start* and extends by increments of *step* up to but not including *end*.

Parameters

- **start** (*Union[float, int], optional*) – The start of the interval. Default: 0 .
- **end** (*Union[float, int], optional*) – The end of the interval, exclusive. Default: None . If None , it defaults to the value of *start*, and 0 is used as the starting value.
- **step** (*Union[float, int], optional*) – The step size with which the array element increments. Default: 1 .

Keyword Arguments

dtype (mindspore.dtype, *optional*) – The required data type of returned Tensor. Default: None . When *dtype* is not specified or None:

If *start*, *end*, and *step* are all integers, the dtype of output is int64,

If *start*, *end*, and *step* contain at least one floating-point number, the dtype of output is float32.

Returns

A 1-D Tensor, cast to *dtype* if provided, may potentially lose precision due to casting.

Raises

- **TypeError** – If *start*, *end* or *step* are not of type int or float.

- **ValueError** –If *step* = 0.
- **ValueError** –If *start* >= *end* when *step* > 0.
- **ValueError** –If *start* <= *end* when *step* < 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> output = mint.arange(1, 6)
>>> print(output)
[1 2 3 4 5]
>>> print(output.dtype)
Int64
>>> output = mint.arange(0, 3, 1.2)
>>> print(output)
[0.  1.2 2.4]
>>> print(output.dtype)
Float32
>>> output = mint.arange(7, 1, -2)
>>> print(output)
[7 5 3]
>>> print(output.dtype)
Int64
>>> output = mint.arange(12, 2, -1, dtype=ms.bfloat16)
>>> print(output)
[12. 11. 10.  9.  8.  7.  6.  5.  4.  3.]
>>> print(output.dtype)
BFloat16
```

mindspore.mint.bernoulli

mindspore.mint.**bernoulli**(*input*, *, *generator=None*)

Sample from the Bernoulli distribution and randomly set the i^{th} element of the *output* to (0 or 1) according to the i^{th} probability value given in the *input*.

$$output_i \sim Bernoulli(p = input_i)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) –The input tensor of Bernoulli distribution, where the i^{th} element 'input_{i}' represents the probability that the corresponding output element 'output_{i}' is set to '1', therefore each element in 'input' have to be in the range '[0,1]'. Supported dtype: float16, float32, float64, bfloat16 (only supported by Atlas A2 training series products).

Keyword Arguments

generator (*mindspore.Generator*, optional) –a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.

Returns

output (Tensor), The output tensor, with the same shape and dtype as *input*.

Raises

- **TypeError** –If dtype of *input* is not one of: float16, float32, float64, bfloat16.
- **ValueError** –If any element of the *input* is not in the range [0, 1].

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> input_x = Tensor(np.ones((3, 3)), mindspore.float32)
>>> output = mint.bernoulli(input_x)
>>> print(output)
[[ 1. 1. 1.]
 [ 1. 1. 1.]
 [ 1. 1. 1.]]
>>> input_x = Tensor(np.zeros((3, 3)), mindspore.float32)
>>> output = mint.bernoulli(input_x)
>>> print(output)
[[ 0. 0. 0.]
 [ 0. 0. 0.]
 [ 0. 0. 0.]]
```

mindspore.mint.bincount

`mindspore.mint.`**`bincount`**(*input*, *weights=None*, *minlength=0*)

Count the occurrences of each value in the input.

If *minlength* is not specified, the length of the output Tensor is the maximum value in the input plus one. If *minlength* is specified, the length of the output Tensor is the maximum value between *minlength* or the maximum value in the input plus one.

Each value in the output Tensor represents the number of occurrences of that index value in the input. If *weights* is specified, the output results are weighted, i.e., $out[n] += weight[i]$ instead of $out[n] += 1$.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) –A one-dimensional Tensor.
- **weights** (`Tensor, optional`) –Weights with the same shape as the input. Default: `None`.
- **minlength** (`int, optional`) –The minimum length of output Tensor. Should be non-negative. Default: `0`.

Returns

Tensor, If input is non-empty, the output shape is $(max(max(input) + 1, minlength),)$, otherwise the shape is $(0,)$.

Raises

- **TypeError** –If *input* or *weights* is not a Tensor.
- **ValueError** –If *input* contains negative values.
- **ValueError** –If *input* is not one-dimensional or *input* and *weights* do not have the same shape.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import mint, Tensor
>>> print(mint.bincount(Tensor(np.arange(5))))
[1 1 1 1 1]
>>> print(mint.bincount(Tensor(np.array([0, 1, 1, 3, 2, 1, 7]))))
[1 3 1 1 0 0 0 1]
>>> w = Tensor(np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6])) # weights
>>> x = Tensor(np.array([0, 1, 1, 2, 2, 2]))
>>> print(mint.bincount(x,  weights=w, minlength=5))
[0.3 0.7 1.1 0.  0. ]
```

mindspore.mint.clone

`mindspore.mint.clone(input)`

Returns a copy of the input tensor.

Warning: This is an experimental API that is subject to change or deletion.

Note: This function is differentiable, and gradients will flow back directly from the calculation result of the function to the *input*.

Parameters

input (Tensor) –A tensor to be copied.

Returns

Tensor, with the same data, shape and type as *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones((3,3)).astype("float32"))
>>> output = mint.clone(input)
>>> print(output)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

mindspore.mint.eye

mindspore.mint.**eye**(*n*, *m=None*, *dtype=None*)

Returns a tensor with ones on the diagonal and zeros in the rest.

Parameters

- **n** (*int*) –The number of rows returned.
- **m** (*int, optional*) –The number of columns returned. If None , the number of columns is as the same as n.
- **dtype** (mindspore.dtype, *optional*) –The data type returned.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.eye(3)
>>> print(output)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>>
>>> output = mindspore.mint.eye(3, 4)
>>> print(output)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
```

mindspore.mint.einsum

mindspore.mint.**einsum**(*equation*, **operands*)

According to the Einstein summation Convention (Einsum), the product of the input tensor elements is summed along the specified dimension. You can use this operator to perform diagonal, reducesum, transpose, matmul, mul, inner product operations, etc.

Note: The sublist format is also supported. For example, einsum_ext(op1, sublist1, op2, sublist2, ···, sublist_out). In this format, equation can be derived by the sublists which are made up of Python's Ellipsis and list of integers in [0, 52). Each operand is followed by a sublist and an output sublist is at the end. Dynamic shape, dynamic rank input is not supported in graph mode (mode=mindspore.GRAPH_MODE).

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **equation** (*str*) −Notation based on the Einstein summation convention, represent the operation you want to do. the value can contain only letters, commas, ellipsis and arrow. The letters(must be in [a-zA-Z]) represent input tensor dimension, commas(,) represent separate tensors, ellipsis indicates the tensor dimension that you do not care about, the left of the arrow indicates the input tensors, and the right of it indicates the desired output dimension. If there are no arrows in the equation, the letters that appear exactly once in the equation will be part of the output, sorted in increasing alphabetical order. The output is computed by multiplying the input operands element-wise, with their dimensions aligned based on the letters, and then summing out the dimensions whose letters are not part of the output. If there is one arrow in the equation, the output letters must appear at least once for some input operand and at most once for the output.
- **operands** (Tensor) −Input tensor used for calculation. The dtype of the tensor must be the same.

Returns

Tensor, the shape of it can be obtained from the *equation* , and the dtype is the same as input tensors.

Raises

- **TypeError** −If *equation* is invalid, or the *equation* does not match the input tensor.
- **ValueError** −If the number in sublist is not in [0, 52) in sublist format.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> equation = "i->"
>>> output = mint.einsum(equation, x)
>>> print(output)
7.0
>>> x = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> y = Tensor(np.array([2.0, 4.0, 3.0]), mindspore.float32)
>>> equation = "i,i->i"
```

(continues on next page)

(continued from previous page)

```
>>> output = mint.einsum(equation, x, y)
>>> print(output)
[ 2. 8. 12.]
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> y = Tensor(np.array([[2.0, 3.0], [1.0, 2.0], [4.0, 5.0]]), mindspore.float32)
>>> equation = "ij,jk->ik"
>>> output = mint.einsum(equation, x, y)
>>> print(output)
[[16. 22.]
 [37. 52.]]
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "ij->ji"
>>> output = mint.einsum(equation, x)
>>> print(output)
[[1. 4.]
 [2. 5.]
 [3. 6.]]
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "ij->j"
>>> output = mint.einsum(equation, x)
>>> print(output)
[5. 7. 9.]
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "...->"
>>> output = mint.einsum(equation, x)
>>> print(output)
21.0
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([2.0, 4.0, 1.0]), mindspore.float32)
>>> equation = "j,i->ji"
>>> output = mint.einsum(equation, x, y)
>>> print(output)
[[ 2. 4. 1.]
 [ 4. 8. 2.]
 [ 6. 12. 3.]]
>>> x = mindspore.Tensor([1, 2, 3, 4], mindspore.float32)
>>> y = mindspore.Tensor([1, 2], mindspore.float32)
>>> output = mint.einsum(x, [..., 1], y, [..., 2], [..., 1, 2])
>>> print(output)
[[1. 2.]
 [2. 4.]
 [3. 6.]
 [4. 8.]]
```

mindspore.mint.empty

mindspore.mint.**empty**(**size*, *dtype=None*, *device=None*) → *Tensor*

Creates a tensor with uninitialized data, whose shape, dtype and device are described by the argument *size*, *dtype* and *device* respectively.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

size (*Union[tuple[int], list[int], int]*) –The specified shape of output tensor. Can be variable numbers of positive integers or tupled or list containing positive integers.

Keyword Arguments

- **dtype** (*mindspore.dtype*, optional) –The specified type of output tensor. If *dtype* is `None` , *mindspore.float32* will be used. Default: `None` .
- **device** (*string, optional*) –The specified device of the output tensor. Support `CPU` and `Ascend`. If *device* = *None*, *mindspore.context.device_target* will be used. Default `None`.

Returns

Tensor, whose dtype and size are defined by input.

Raises

TypeError –If *size* is neither an int nor a tuple or list of int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> output = mint.empty((2, 3), dtype=mindspore.float32)
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.mint.empty_like

`mindspore.mint.`**empty_like**(*input*, *, *dtype=None*, *device=None*)

Returns an uninitialized Tensor with the same shape as the *input*. Its dtype is specified by *dtype* and its device is specified by *device*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (`Tensor`) –Tensor of any dimension.

Keyword Arguments

- **dtype** (*mindspore.dtype*, optional) –The specified dtype of the output tensor. If *dtype* = *None*, the tensor will have the same dtype as input *input*. Default `None`.
- **device** (*string, optional*) –The specified device of the output tensor. Support `CPU` and `Ascend`. If *device* = *None*, the tensor will have the same device as input *input* and if the device of the input tensor is not defined, the value set by *mindspore.set_device()* will be used. Default `None`.

Returns

Tensor, has the same shape, type and device as *input* but with uninitialized data (May be a random value).

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import mint, Tensor
>>> x = Tensor([[1, 2, 3], [4, 5, 6]])
>>> output1 = mint.empty_like(x)
>>> print(output1)
[[0 0 0]
 [0 0 0]]
>>> output2 = mint.empty_like(x, dtype=mindspore.float64)
>>> print(output2)
[[0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.mint.full

mindspore.mint.**full**(*size*, *fill_value*, *, *dtype=None*)

Create a Tensor of the specified shape and fill it with the specified value.

Parameters

- **size** (*Union(tuple[int], list[int])*) – The specified shape of output tensor.
- **fill_value** (*Union(number.Number, Tensor)*) – Value to fill the returned tensor. It can be a Scalar number, a 0-D Tensor, or a 1-D Tensor with only one element.

Keyword Arguments

dtype (mindspore.dtype, *optional*) – The specified type of output tensor. *bool_* and *number* are supported, for details, please refer to *mindspore.dtype* . Default: None .

Returns

Tensor.

Raises

- **TypeError** – If *size* is not a tuple or list.
- **ValueError** – The element in *size* is less than 0.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint
>>> output = mint.full((2, 2), 1)
>>> print(output)
[[1. 1.]
 [1. 1.]]
>>> output = mint.full((3, 3), 0)
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.mint.full_like

mindspore.mint.**full_like**(*input*, *fill_value*, *, *dtype=None*)

Return a Tensor of the same shape as *input* and filled with *fill_value*.

Parameters

- **input** (Tensor) – input Tensor and the output Tensor have the same shape as *input*.
- **fill_value** (*Number*) – Value to fill the returned tensor. Complex numbers are not supported for now.

Keyword Arguments

dtype (mindspore.dtype, *optional*) – The specified type of output tensor. *bool_* and *number* are supported, for details, please refer to *mindspore.dtype* . Default: None .

Returns

Tensor.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[0, 1], [2, 1]], dtype=mindspore.int32)
>>> output = mint.full_like(input, 1)
>>> print(output)
[[1 1]
 [1 1]]
>>> input = Tensor([[0, 1, 1], [2, 1, 2], [1, 3, 4]], dtype=mindspore.int32)
>>> output = mint.full_like(input, 0, dtype=mindspore.float32)
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.mint.linspace

mindspore.mint.**linspace**(*start*, *end*, *steps*, *, *dtype=None*)

Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].

$$step = (end - start)/(steps - 1)$$
$$output = [start, start + step, start + 2 * step, ..., end]$$

Warning: Atlas training series does not support int16 dtype currently.

Parameters

- **start** (*Union[float, int]*) –Start value of interval.
- **end** (*Union[float, int]*) –Last value of interval.
- **steps** (*int*) –Number of elements.

Keyword Arguments

dtype (mindspore.dtype, *optional*) –The data type returned.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.linspace(3, 10, 5)
>>> print(output)
[ 3.    4.75  6.5   8.25 10.  ]
>>> output = mindspore.mint.linspace(-10, 10, 5)
>>> print(output)
[-10.  -5.   0.   5.  10.]
>>> output = mindspore.mint.linspace(-10, 10, 1)
>>> print(output)
[-10.]
```

mindspore.mint.ones

mindspore.mint.**ones**(*size*, *, *dtype=None*)

Creates a tensor filled with value ones.

Creates a tensor with shape described by the first argument and fills it with value ones in type of the second argument.

Parameters

size (*Union[tuple[int], list[int], int, Tensor]*) –The specified shape of output tensor. Only positive integer or tuple or Tensor containing positive integers are allowed. If it is a Tensor, it must be a 0-D or 1-D Tensor with int32 or int64 dtypes.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The specified type of output tensor. If *dtype* is `None` , *mindspore.float32* will be used. Default: `None` .

Returns

Tensor, whose dtype and size are defined by input.

Raises

TypeError – If *size* is neither an int nor a tuple/list/Tensor of int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> output = mint.ones((2, 2), dtype=mindspore.float32)
>>> print(output)
[[1. 1.]
 [1. 1.]]
```

mindspore.mint.ones_like

mindspore.mint.**ones_like**(*input*, *, *dtype=None*)

Creates a tensor filled with 1, with the same shape as input, and its data type is determined by the given dtype.

If *dtype* = *None*, the tensor will have the same dtype as input *input*.

Parameters

input (`Tensor`) – Tensor of any dimension.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The specified dtype of the output tensor. If *dtype* is `None` , the dtype of the input tensor will be used. Default: `None` .

Returns

Tensor, has the same shape as *input* but filled with ones.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.int32))
>>> output = mint.ones_like(x)
>>> print(output)
[[1 1]
 [1 1]]
```

mindspore.mint.randint

mindspore.mint.**randint**(*low=0*, *high*, *size*, ***, *generator=None*, *dtype=None*) → *Tensor*

Returns a new tensor filled with integer numbers from the uniform distribution over an interval $[low, high)$ based on the given shape and dtype.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **low** (*int, optional*) – the lower bound of the generated random number. Default: 0.
- **high** (*int*) – the upper bound of the generated random number
- **size** (*Union[tuple(int), list(int)]*) – Shape of the new tensor, e.g. $(2, 3)$.

Keyword Arguments

- **generator** (*mindspore.Generator*, optional) – a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.
- **dtype** (*mindspore.dtype*, optional) – Designated tensor dtype. If None, *mindspore.int64* will be applied. Default: None .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the uniform distribution on the interval $[low, high)$.

Raises

- **TypeError** – If *size* is not a tuple.
- **TypeError** – If *low* or *high* is not integer.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint
>>> print(mint.randint(0, 5, (2, 3)).shape)
(2, 3)
```

mindspore.mint.randint_like

mindspore.mint.**randint_like**(*input*, *low=0*, *high*, ***, *dtype=None*) → *Tensor*

Returns a new tensor filled with integer numbers from the uniform distribution over an interval $[low, high)$ based on the given dtype and shape of the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −Input Tensor to specify the output shape and its default dtype.
- **low** (*int, optional*) −the lower bound of the generated random number. Default: 0.
- **high** (*int*) −the upper bound of the generated random number

Keyword Arguments

dtype (*mindspore.dtype*, optional) −Designated tensor dtype. If None, the same dtype of *input* will be applied. Default: None .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the uniform distribution on the interval $[low, high)$.

Raises

TypeError −If *low* or *high* is not integer.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> a = Tensor([[2, 3, 4], [1, 2, 3]])
>>> low = 0
>>> high = 5
>>> print(mint.randint_like(a, low, high, dtype=ms.int32).shape)
(2, 3)
```

mindspore.mint.randn

mindspore.mint.**randn**(**size*, *generator=None*, *dtype=None*)

Returns a new tensor filled with numbers from the normal distribution over an interval $[0, 1)$ based on the given shape and dtype.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

size (*Union[int, tuple(int), list(int)]*) –Shape of the new tensor, e.g. $(2, 3)$ or 2.

Keyword Arguments

- **generator** (*mindspore.Generator*, optional) –a pseudorandom number generator. Default: `None`, uses the default pseudorandom number generator.
- **dtype** (*mindspore.dtype*, optional) –Designated tensor dtype. If None, *mindspore.float32* will be applied. Default: `None` .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the normal distribution on the interval $[0, 1)$.

Raises

ValueError –If *size* contains negative numbers.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import mint
>>> print(mint.randn(2, 3).shape)
(2, 3)
```

mindspore.mint.randn_like

mindspore.mint.**randn_like**(*input*, ***, *dtype=None*)

Returns a new tensor filled with numbers from the normal distribution over an interval $[0, 1)$ based on the given dtype and shape of the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (`Tensor`) –Input Tensor to specify the output shape and its default dtype.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –Designated Tensor dtype, it must be float type. If `None`, the same dtype of *input* will be applied. Default: `None` .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the normal distribution on the interval $[0, 1)$.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> a = Tensor([[2, 3, 4], [1, 2, 3]])
>>> print(mint.randn_like(a, dtype=ms.float32).shape)
(2, 3)
```

mindspore.mint.randperm

`mindspore.mint.`**`randperm`**(*n*, ***, *generator=None*, *dtype=mstype.int64*)

Generates random permutation of integers from 0 to n-1.

Warning:

- This is an experimental API that is subject to change or deletion.

Parameters

n (*Union[Tensor, int]*) – size of the permutation. int or Tensor with shape: () or (1,) and data type int64. The value of *n* must be greater than zero.

Keyword Arguments

- **generator** (*mindspore.Generator*, optional) – a pseudorandom number generator. Default: `None`, uses the default pseudorandom number generator.
- **dtype** (*mindspore.dtype, optional*) – The type of output. Default: mstype.int64.

Returns

Tensor with shape (n,) and type *dtype*.

Raises

- **TypeError** – If *dtype* is not supported.
- **ValueError** – If *n* is a negative or 0 element.
- **ValueError** – If *n* is larger than the maximal data of the set dtype.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import mint
>>> from mindspore import dtype as mstype
>>> n = 4
>>> output = mint.randperm(n, dtype=mstype.int64)
>>> print(output.shape)
(4,)
```

mindspore.mint.zeros

mindspore.mint.**zeros**(*size*, *, *dtype=None*)

Creates a tensor filled with 0 with shape described by *size* and fills it with value 0 in type of *dtype*.

Parameters

size (*Union[tuple[int], list[int], int, Tensor]*) – The specified shape of output tensor. Only positive integer or tuple or Tensor containing positive integers are allowed. If it is a Tensor, it must be a 0-D or 1-D Tensor with int32 or int64 dtypes.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The specified type of output tensor. If *dtype* is `None` , mindspore.float32 will be used. Default: `None` .

Returns

Tensor, whose dtype and size are defined by input.

Raises

TypeError – If *size* is neither an int nor a tuple/list/Tensor of int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> output = mint.zeros((2, 2), dtype=mindspore.float32)
>>> print(output)
[[0. 0.]
 [0. 0.]]
```

mindspore.mint.zeros_like

mindspore.mint.**zeros_like**(*input*, *, *dtype=None*)

Creates a tensor filled with 0, with the same size as input. Its data type is determined by the given dtype.

If *dtype* = *None*, the tensor will have the same dtype as input *input*.

Parameters

input (Tensor) –Tensor of any dimension.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The specified dtype of the output tensor. If *dtype* is None , the dtype of the input tensor will be used. Default: None .

Returns

Tensor, filled with 0.

Raises

TypeError –If dtype is not a MindSpore dtype.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.arange(4).reshape(2, 2))
>>> output = mint.zeros_like(x, dtype=mindspore.float32)
>>> print(output)
[[0. 0.]
 [0. 0.]]
```

4.1.2 Indexing, Slicing, Joining, Mutating Operations

API Name	Description	Supported Platforms
mindspore.mint.cat	Connect input tensors along with the given dimension.	Ascend
mindspore.mint.chunk	Cut the input Tensor into *chunks* sub-tensors along the specified axis.	Ascend
mindspore.mint.concat	Alias for *mindspore.mint.cat()*.	Ascend
mindspore.mint.count_nonzero	Count the number of non-zero elements in the Tensor *input* on a given dimension *dim*.	Ascend
mindspore.mint.gather	Gather data from a tensor by indices.	Ascend
mindspore.mint.index_add	Accumulate the elements of *alpha* times *source* into the *input* by adding to the index in the order given in *index*.	Ascend

continues on next page

Table 2 – continued from previous page

mindspore.mint.index_select	Generates a new Tensor that accesses the values of *input* along the specified *dim* dimension using the indices specified in *index*.	Ascend
mindspore.mint.masked_select	Return a new 1-D tensor which indexes the *input* tensor according to the boolean *mask*.	Ascend
mindspore.mint.permute	Permutes the dimensions of the input tensor according to input *dims* .	Ascend
mindspore.mint.reshape	Reshape the input tensor based on the given shape.	Ascend
mindspore.mint.scatter	Update the value in *src* to *input* according to the specified index.	Ascend
mindspore.mint.scatter_add	Add all elements in *src* to the index specified by *index* to *input* along dimension specified by *dim*.	Ascend
mindspore.mint.split	Splits the Tensor into chunks along the given dim.	Ascend
mindspore.mint.narrow	Obtains a tensor of a specified length at a specified start position along a specified axis.	Ascend
mindspore.mint.nonzero	Return the positions of all non-zero values.	Ascend
mindspore.mint.tile	Creates a new tensor by repeating the elements in the input tensor *dims* times.	Ascend
mindspore.mint.tril	Zero the input tensor above the diagonal specified.	Ascend
mindspore.mint.select	Slices the input tensor along the selected dimension at the given index.	Ascend
mindspore.mint.squeeze	Return the Tensor after deleting the dimension of size 1 in the specified *dim*.	Ascend
mindspore.mint.stack	Stacks a list of tensors in specified dim.	Ascend
mindspore.mint.swapaxes	Alias for mindspore.mint.transpose() .	Ascend
mindspore.mint.transpose	Interchange two axes of a tensor.	Ascend
mindspore.mint.triu	Zero the input tensor below the diagonal specified.	Ascend
mindspore.mint.unbind	Unbind a tensor dimension in specified axis.	Ascend
mindspore.mint.unique_consecutive	Returns the elements that are unique in each consecutive group of equivalent elements in the input tensor.	Ascend
mindspore.mint.unsqueeze	Adds an additional dimension to the input tensor at the given dimension.	Ascend
mindspore.mint.where	Selects elements from *input* or *other* based on *condition* and returns a tensor.	Ascend

mindspore.mint.cat

mindspore.mint.**cat**(*tensors*, *dim=0*)

Connect input tensors along with the given dimension.

The input data is a tuple or a list of tensors. These tensors have the same rank R. Set the given dimension as m, and $0 \le m < R$. Set the number of input tensors as N. For the i-th tensor t_i, it has the shape of $(x_1, x_2, ..., x_{mi}, ..., x_R)$. x_{mi} is the m-th dimension of the t_i. Then, the shape of the output tensor is

$$(x_1, x_2, ..., \sum_{i=1}^{N} x_{mi}, ..., x_R)$$

Parameters

- **tensors** (*Union[tuple, list]*) −A tuple or a list of input tensors. Suppose there are two tensors in this tuple or list, namely t1 and t2. To perform *concat* in the dimension 0 direction, except for the 0-th dimension, all other

dimensions should be equal, that is, $t1.shape[1] = t2.shape[1], t1.shape[2] = t2.shape[2], ..., t1.shape[R-1] = t2.shape[R-1]$, where R represents the rank of tensor.

- **dim** (*int, optional*) –The specified dimension, whose value is in range $[-R, R)$. Default: `0` .

Returns

Tensor, the shape is $(x_1, x_2, ..., \sum_{i=1}^{N} x_{mi}, ..., x_R)$. The data type is the same with *tensors*.

Raises

- **TypeError** –If *dim* is not an int.
- **ValueError** –If *tensors* have different dimension of tensor.
- **ValueError** –If *dim* not in range $[-R, R)$.
- **ValueError** –If tensor's shape in *tensors* except for *dim* are different.
- **ValueError** –If *tensors* is an empty tuple or list.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> input_x1 = Tensor(np.array([[0, 1], [2, 1]]).astype(np.float32))
>>> input_x2 = Tensor(np.array([[0, 1], [2, 1]]).astype(np.float32))
>>> output = mint.cat((input_x1, input_x2))
>>> print(output)
[[0. 1.]
 [2. 1.]
 [0. 1.]
 [2. 1.]]
>>> output = mint.cat((input_x1, input_x2), 1)
>>> print(output)
[[0. 1. 0. 1.]
 [2. 1. 2. 1.]]
```

mindspore.mint.chunk

mindspore.mint.**chunk**(*input*, *chunks*, *dim=0*)

Cut the input Tensor into *chunks* sub-tensors along the specified axis.

Note: This function may return less than the specified number of chunks!

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –A Tensor to be cut.
- **chunks** (*int*) –Number of sub-tensors to cut.
- **dim** (*int, optional*) –Specify the dimensions that you want to split. Default: 0 .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –If argument *input* is not Tensor.
- **TypeError** –The sum of *chunks* is not int.
- **TypeError** –If argument *dim* is not int.
- **ValueError** –If argument *dim* is out of range of $[-input.ndim, input.ndim)$.
- **ValueError** –If argument *chunks* is not positive number.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> input_x = np.arange(9).astype("float32")
>>> output = mindspore.mint.chunk(Tensor(input_x), 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

mindspore.mint.concat

mindspore.mint.**concat**(*tensors*, *dim=0*)

Alias for *mindspore.mint.cat()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Supported Platforms:

Ascend

mindspore.mint.count_nonzero

mindspore.mint.**count_nonzero**(*input*, *dim=None*)

Count the number of non-zero elements in the Tensor *input* on a given dimension *dim*. If no dim is specified then all non-zeros in the tensor are counted.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Input data is used to count non-zero numbers. With shape $(*)$ where $*$ means, any number of additional dimensions.
- **dim** (*Union[None, int, tuple(int), list(int)], optional*) –Count the dimension of the number of non-zero elements. Default value: None, which indicates that the number of non-zero elements is calculated. If *dim* is None, all elements in the tensor are summed up.

Returns

Tensor, number of nonzero element across dim specified by *dim*.

Raises

- **TypeError** –If *input* is not tensor.
- **TypeError** –If *dim* is not int, tuple(int), list(int) or None.
- **ValueError** –If any value in *dim* is not in range $[-input.ndim, input.ndim)$.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> import mindspore
>>> # case 1: each value specified.
>>> x = Tensor(np.array([[0, 1, 0], [1, 1, 0]]).astype(np.float32))
>>> nonzero_num = mint.count_nonzero(input=x, dim=[0, 1])
>>> print(nonzero_num)
3
>>> # case 2: all value is default.
>>> nonzero_num = mint.count_nonzero(input=x)
>>> print(nonzero_num)
3
>>> # case 3: dim value was specified 0.
>>> nonzero_num = mint.count_nonzero(input=x, dim=[0,])
>>> print(nonzero_num)
[1 2 0]
>>> # case 4: dim value was specified 1.
>>> nonzero_num = mint.count_nonzero(input=x, dim=[1,])
>>> print(nonzero_num)
[1 2]
```

mindspore.mint.gather

mindspore.mint.**gather**(*input*, *dim*, *index*)

Gather data from a tensor by indices.

$$output[(i_0, i_1, ..., i_{dim}, i_{dim+1}, ..., i_n)] = input[(i_0, i_1, ..., index[(i_0, i_1, ..., i_{dim}, i_{dim+1}, ..., i_n)], i_{dim+1}, ..., i_n)]$$

Warning: On Ascend, the behavior is unpredictable in the following cases:

- the value of *index* is not in the range *[-input.shape[dim], input.shape[dim])* in forward;
- the value of *index* is not in the range *[0, input.shape[dim])* in backward.

Parameters

- **input** (Tensor) –The target tensor to gather values.
- **dim** (*int*) –the axis to index along, must be in range *[-input.rank, input.rank)*.
- **index** (Tensor) –The index tensor, with int32 or int64 data type. A valid *index* should be:
  - *index.rank == input.rank*;
  - for *axis != dim*, *index.shape[axis] <= input.shape[axis]*;
  - the value of *index* is in range *[-input.shape[dim], input.shape[dim])*.

Returns

Tensor, has the same type as *input* and the same shape as *index*.

Raises

- **ValueError** –If the shape of *index* is illegal.
- **ValueError** –If *dim* is not in *[-input.rank, input.rank)*.
- **ValueError** –If the value of *index* is out of the valid range.
- **TypeError** –If the type of *index* is illegal.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> index = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> output = mint.gather(input, 1, index)
>>> print(output)
[[-0.1 -0.1]
 [0.5   0.5]]
```

mindspore.mint.index_add

mindspore.mint.**index_add**(*input*, *dim*, *index*, *source*, *alpha=1*)

Accumulate the elements of *alpha* times *source* into the *input* by adding to the index in the order given in *index*. For example, if `dim == 0`, `index[i] == j`, and `alpha = -1`, then the *i* th row of *source* is subtracted from the *j* th row of *input*. The *dim* th dimension of *source* must have the same size as the length of *index*, and all other dimensions must match *input*, or an error will be raised. For a 3-D tensor, the output is defined as follows:

$$
\begin{aligned}
input[index[i],\ :,\ :] &+= alpha * source[i,\ :,\ :] \quad \#if\ dim == 0 \\
input[:,\ index[i],\ :] &+= alpha * source[:,\ i,\ :] \quad \#if\ dim == 1 \\
input[:,\ :,\ index[i]] &+= alpha * source[:,\ :,\ i] \quad \#if\ dim == 2
\end{aligned}
$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input Tensor.
- **dim** (*int*) –The dimension along which to index.
- **index** (Tensor) –Add the value of "input Tensor" and *source* along the dimension of the *dim* according to the specified index value, with data type int32. The *index* must be 1D with the same size as the size of *source* in the *dim* dimension. The values of *index* should be in [0, b), where the b is the size of "input Tensor" in the *dim* dimension.
- **source** (Tensor) –The input tensor with the value to add. Must have same data type as "input Tensor". The shape must be the same as "input Tensor" except the *dim* th dimension.
- **alpha** (*number, optional*) –The scalar multiplier for source. Default: `1`.

Returns

Tensor, has the same shape and dtype as *input*.

Raises

- **TypeError** –If neither *index* nor *source* is a Tensor.
- **ValueError** –If the value of *dim* is out of the dimension range of *source* shape.
- **ValueError** –If *index* rank is not the same as *source* rank.
- **ValueError** –If shape of *index* is not 1D or size of *index* is not equal to dimension of source[dim].
- **ValueError** –If the shape of *source* is not the same as that of *input* except the *dim* axis.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> index = Tensor(np.array([0, 2]), mindspore.int32)
>>> y = Tensor(np.array([[0.5, 1.0], [1.0, 1.5], [2.0, 2.5]]), mindspore.float32)
>>> output = mint.index_add(x, 1, index, y, alpha=1)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[ 1.5  2.   4. ]
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
```

mindspore.mint.index_select

mindspore.mint.**index_select**(*input*, *dim*, *index*)

Generates a new Tensor that accesses the values of *input* along the specified *dim* dimension using the indices specified in *index*. The new Tensor has the same number of dimensions as *input*, with the size of the *dim* dimension being equal to the length of *index*, and the size of all other dimensions will be unchanged from the original *input* Tensor.

Note: The value of index must be in the range of *[0, input.shape[dim])*, the result is undefined out of range.

Parameters

- **input** (Tensor) –The input Tensor.
- **dim** (*int*) –The dimension to be indexed.
- **index** (Tensor) –A 1-D Tensor with the indices.

Returns

Tensor, has the same dtype as input Tensor.

Raises

- **TypeError** –If *input* or *index* is not a Tensor.
- **TypeError** –If *dim* is not int number.
- **ValueError** –If the value of *dim* is out the range of *[-input.ndim, input.ndim - 1]*.
- **ValueError** –If the dimension of *index* is not equal to 1.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.arange(16).astype(np.float32).reshape(2, 2, 4))
>>> print(input)
[[[ 0.  1.  2.  3.]
[ 4.  5.  6.  7.]]
[[ 8.  9. 10. 11.]
[12. 13. 14. 15.]]]
>>> index = Tensor([0,], mindspore.int32)
>>> y = mint.index_select(input, 1, index)
>>> print(y)
```

(continues on next page)

(continued from previous page)

```
[[[ 0.  1.  2.  3.]]
[[ 8.  9. 10. 11.]]]
```

mindspore.mint.masked_select

mindspore.mint.**masked_select**(*input*, *mask*)

Return a new 1-D tensor which indexes the *input* tensor according to the boolean *mask*.

Support broadcast.

Parameters

- **input** (Tensor) – The input tensor.
- **mask** (Tensor[bool]) – The input mask.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1, 2, 3, 4], mindspore.int64)
>>> mask = mindspore.tensor([1, 0, 1, 0], mindspore.bool_)
>>> output = mindspore.mint.masked_select(x, mask)
>>> print(output)
[1 3]
```

mindspore.mint.permute

mindspore.mint.**permute**(*input*, *dims*)

Permutes the dimensions of the input tensor according to input *dims* .

Parameters

- **input** (Tensor) – Input Tensor.
- **dims** (tuple(int)) – The order of the dimensions. Permute rearranges the *input* according to the order of the *dims*.

Returns

Tensor, has the same dimension as input tensor, with *axis* suitably permuted.

Raises

- **ValueError** – If *dims* is None.
- **ValueError** – If the number of elements of *dims* is not equal to *input* ndim.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]]),␣
↪mindspore.float32)
>>> input_perm = (0, 2, 1)
>>> print(mint.permute(input_x, input_perm))
[[[ 1.  4.]
  [ 2.  5.]
  [ 3.  6.]]
 [[ 7. 10.]
  [ 8. 11.]
  [ 9. 12.]]]
```

mindspore.mint.reshape

mindspore.mint.**reshape**(*input*, *shape*)

Reshape the input tensor based on the given shape.

Note: The -1 in the parameter *shape* indicates that the size of that dimension is inferred from the other dimensions and the total number of elements in input tensor.

Parameters

- **input** (Tensor) –The input tensor.
- **shape** (*Union[tuple[int], list[int], Tensor[int]]*) –New shape.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]], mindspore.float32)
>>> # case1: Parameter `shape` does not contain -1.
>>> output = mindspore.mint.reshape(input, (3, 2))
>>> print(output)
[[-0.1  0.3]
 [ 3.6  0.4]
 [ 0.5 -3.2]]
>>> # case2: Parameter `shape` contains -1.
>>> output = mindspore.mint.reshape(input, (-1, 6))
>>> print(output)
[[-0.1  0.3  3.6  0.4  0.5 -3.2]]
```

mindspore.mint.scatter

mindspore.mint.**scatter**(*input*, *dim*, *index*, *src*)

Update the value in *src* to *input* according to the specified index. For a 3-D tensor, the output will be:

```
output[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0

output[i][index[i][j][k]][k] = src[i][j][k]  # if dim == 1

output[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

Note: The backward is supported only for the case *src.shape* == *index.shape* when *src* is a tensor.

Parameters

- **input** (Tensor) −The target tensor. The rank of *input* must be at least 1.
- **dim** (*int*) −Which axis to scatter. Accepted range is [-r, r) where r = rank(input).
- **index** (Tensor) −The index to do update operation whose data must be positive number with type of mindspore.int32 or mindspore.int64. Same rank as *input* . And accepted range is [-s, s) where s is the size along axis.
- **src** (Tensor, *float*) −The data doing the update operation with *input*. Can be a tensor with the same data type as *input* or a float number to scatter.

Returns

Tensor, has the same shape and type as *input* .

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If rank of any of *input* , *index* and *src* is less than 1.
- **ValueError** −If the rank of *src* is not equal to the rank of *input* .
- **TypeError** −If the data types of *input* and *src* have different dtypes.
- **RuntimeError** −If *index* has negative elements.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = mint.scatter(input=input, dim=1, index=index, src=src)
>>> print(out)
[[1. 2. 8. 4. 8.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
```

(continues on next page)

(continued from previous page)

```
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = mint.scatter(input=input, dim=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
[0. 0. 0. 0. 0.]
[4. 5. 6. 0. 0.]
[0. 0. 0. 0. 0.]
[7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = mint.scatter(input=input, dim=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
[4. 0. 5. 0. 6.]
[7. 0. 8. 0. 9.]
[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]]
```

mindspore.mint.scatter_add

mindspore.mint.**scatter_add**(*input*, *dim*, *index*, *src*)

Add all elements in *src* to the index specified by *index* to *input* along dimension specified by *dim*. It takes three inputs *input*, *src* and *index* of the same rank r >= 1.

For a 3-D tensor, the operation updates input as follows:

```
input[index[i][j][k]][j][k] += src[i][j][k]  # if dim == 0

input[i][index[i][j][k]][k] += src[i][j][k]  # if dim == 1

input[i][j][index[i][j][k]] += src[i][j][k]  # if dim == 2
```

Parameters

- **input** (Tensor) −The target tensor. The rank must be at least 1.
- **dim** (*int*) −Which dim to scatter. Accepted range is [-r, r) where r = rank(*input*).
- **index** (Tensor) −The index of *input* to do scatter operation whose data type must be mindspore.int32 or mindspore.int64. Same rank as *input*. Except for the dimension specified by *dim*, the size of each dimension of *index* must be less than or equal to the size of the corresponding dimension of *input*.
- **src** (Tensor) −The tensor doing the scatter operation with *input*, has the same type as *input* and the size of each dimension must be greater than or equal to that of *index*.

Returns

Tensor, has the same shape and type as *input*.

Raises

- **TypeError** −If *index* is neither int32 nor int64.
- **ValueError** −If anyone of the rank among *input*, *index* and *src* is less than 1.
- **ValueError** −If the rank of *input*, *index* and *src* is not the same.

- **ValueError** –The size of any dimension of *index* except the dimension specified by *dim* is greater than the size of the corresponding dimension of *input*.
- **ValueError** –If the size of any dimension of *src* is less than that of *index*.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[1, 2, 3, 4, 5]]), dtype=ms.float32)
>>> src = Tensor(np.array([[8, 8]]), dtype=ms.float32)
>>> index = Tensor(np.array([[2, 4]]), dtype=ms.int64)
>>> out = mint.scatter_add(input=input, dim=1, index=index, src=src)
>>> print(out)
[[1. 2. 11. 4. 13.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 0, 0], [2, 2, 2], [4, 4, 4]]), dtype=ms.int64)
>>> out = mint.scatter_add(input=input, dim=0, index=index, src=src)
>>> print(out)
[[1. 2. 3. 0. 0.]
 [0. 0. 0. 0. 0.]
 [4. 5. 6. 0. 0.]
 [0. 0. 0. 0. 0.]
 [7. 8. 9. 0. 0.]]
>>> input = Tensor(np.zeros((5, 5)), dtype=ms.float32)
>>> src = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), dtype=ms.float32)
>>> index = Tensor(np.array([[0, 2, 4], [0, 2, 4], [0, 2, 4]]), dtype=ms.int64)
>>> out = mint.scatter_add(input=input, dim=1, index=index, src=src)
>>> print(out)
[[1. 0. 2. 0. 3.]
 [4. 0. 5. 0. 6.]
 [7. 0. 8. 0. 9.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

mindspore.mint.split

mindspore.mint.**split**(*tensor*, *split_size_or_sections*, *dim=0*)

Splits the Tensor into chunks along the given dim.

Parameters

- **tensor** (Tensor) –A Tensor to be divided.
- **split_size_or_sections** (*Union[int, tuple(int), list(int)]*) –If *split_size_or_sections* is an int type, *tensor* will be split into equally sized chunks, each chunk with size *split_size_or_sections*. Last chunk will be smaller than *split_size_or_sections* if *tensor.shape[dim]* is not divisible by *split_size_or_sections*. If *split_size_or_sections* is a list type, then *tensor* will be split into len(split_size_or_sections) chunks with sizes *split_size_or_sections* along the given *dim*.
- **dim** (*int, optional*) –The dim along which to split. Default: `0` .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –If argument *tensor* is not Tensor.
- **TypeError** –If argument *dim* is not int.
- **ValueError** –If argument *dim* is out of range of [-tensor.ndim, tensor.ndim).
- **TypeError** –If each element in *split_size_or_sections* is not integer.
- **TypeError** –If argument *split_size_or_sections* is not int, tuple(int) or list(int).
- **ValueError** –The sum of *split_size_or_sections* is not equal to tensor.shape[dim].

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import mint, Tensor
>>> input_x = np.arange(9).astype("float32")
>>> output = mint.split(Tensor(input_x), 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  1.00000000e+00,  2.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 3.00000000e+00,  4.00000000e+00,  5.
↪00000000e+00]),
 Tensor(shape=[3], dtype=Float32, value= [ 6.00000000e+00,  7.00000000e+00,  8.
↪00000000e+00]))
```

mindspore.mint.narrow

mindspore.mint.**narrow**(*input*, *dim*, *start*, *length*)

Obtains a tensor of a specified length at a specified start position along a specified axis.

Parameters

- **input** (Tensor) –the tensor to narrow.
- **dim** (*int*) –the axis along which to narrow.
- **start** (*Union[int, Tensor[int]]*) –the starting dimension.
- **length** (*int*) –the distance to the ending dimension.

Returns

output (Tensors) - The narrowed tensor.

Raises

- **ValueError** –the rank of *input* is 0.
- **ValueError** –the value of *dim* is out the range [-input.ndim, input.ndim).
- **ValueError** –the value of *start* is out the range [-input.shape[dim], input.shape[dim]].

- **ValueError** –the value of *length* is out the range [0, input.shape[dim]-start].

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> x = Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], mindspore.int32)
>>> output = mint.narrow(x, 0, 0, 2)
>>> print(output)
[[ 1 2 3]
 [ 4 5 6]]
>>> output = mint.narrow(x, 1, 1, 2)
>>> print(output)
[[ 2 3]
 [ 5 6]
 [ 8 9]]
```

mindspore.mint.nonzero

mindspore.mint.**nonzero**(*input*, *, *as_tuple=False*)

Return the positions of all non-zero values.

Parameters

input (Tensor) –The input tensor.

Note:

- Ascend: Rank of Input tensor can be equal to 0 except jit level O2 mode.

Keyword Arguments

as_tuple (bool, *optional*) –Whether the output is tuple. Default False .

Returns

Tensor or tuple of tensors

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([[[1,  0], [-5, 0]]], mindspore.int32)
>>> output = mindspore.mint.nonzero(x)
>>> print(output)
[[0 0 0]
 [0 1 0]]
>>> x = mindspore.tensor([1, 0, 2, 0, 3], mindspore.int32)
>>> output = mindspore.mint.nonzero(x, as_tuple=False)
>>> print(output)
[[0]
 [2]
 [4]]
>>> x = mindspore.tensor([[[1,  0], [-5, 0]]], mindspore.int32)
>>> output = mindspore.mint.nonzero(x, as_tuple=True)
>>> print(output)
(Tensor(shape=[2], dtype=Int64, value=[0, 0]),
 Tensor(shape=[2], dtype=Int64, value=[0, 1]),
 Tensor(shape=[2], dtype=Int64, value=[0, 0]))
>>> x = mindspore.tensor([1, 0, 2, 0, 3], mindspore.int32)
>>> output = mindspore.mint.nonzero(x, as_tuple=True)
>>> print(output)
(Tensor(shape=[3], dtype=Int64, value=[0, 2, 4]), )
```

mindspore.mint.tile

mindspore.mint.**tile**(*input*, *dims*)

Creates a new tensor by repeating the elements in the input tensor *dims* times.

The i'th dimension of output tensor has *input.shape[i] * dims[i]* elements, and the values of *input* are repeated *dims[i]* times along the i'th dimension.

Note:

- On Ascend, the number of *dims* should not exceed 8, and currently does not support scenarios where more than 4 dimensions are repeated simultaneously.
- If *input.dim* = d, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * y_1, x_2 * y_2, ..., x_S * y_S)$.
- If *input.dim* < d, prepend 1 to the shape of *input* until their lengths are consistent. Such as set the shape of *input* as $(1, ..., x_1, x_2, ..., x_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(1 * y_1, ..., x_R * y_R, x_S * y_S)$.
- If *input.dim* > d, prepend 1 to *dims* until their lengths are consistent. Such as set the *dims* as $(1, ..., y_1, y_2, ..., y_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * 1, ..., x_R * y_R, x_S * y_S)$.

Parameters

- **input** (Tensor) –The input tensor.
- **dims** (*tuple[int]*) –The specified number of repetitions in each dimension.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2], [3, 4]])
>>> mindspore.mint.tile(input, (2, 3))
Tensor(shape=[4, 6], dtype=Int64, value=
[[1, 2, 1, 2, 1, 2],
 [3, 4, 3, 4, 3, 4],
 [1, 2, 1, 2, 1, 2],
 [3, 4, 3, 4, 3, 4]])
>>> mindspore.mint.tile(input, (2, 3, 2))
Tensor(shape=[2, 6, 4], dtype=Int64, value=
[[[1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4]],
 [[1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4],
  [1, 2, 1, 2],
  [3, 4, 3, 4]]])
```

mindspore.mint.tril

mindspore.mint.**tril**(*input*, *diagonal=0*)

Zero the input tensor above the diagonal specified.

Parameters

- **input** (Tensor) –The input tensor. The rank must be at least 2.
- **diagonal** (*int, optional*) –The diagonal specified of 2-D tensor. Default 0 represents the main diagonal.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[ 1,  2,  3,  4],
...                           [ 5,  6,  7,  8],
...                           [10, 11, 12, 13],
...                           [14, 15, 16, 17]])
>>> mindspore.mint.tril(input)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  0,  0,  0],
 [ 5,  6,  0,  0],
 [10, 11, 12,  0],
 [14, 15, 16, 17]])
>>> mindspore.mint.tril(input, 1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  0,  0],
 [ 5,  6,  7,  0],
 [10, 11, 12, 13],
 [14, 15, 16, 17]])
>>> mindspore.mint.tril(input, -1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 0,  0,  0,  0],
 [ 5,  0,  0,  0],
 [10, 11,  0,  0],
 [14, 15, 16,  0]])
>>> input = mindspore.tensor([[[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]],
...                           [[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]]])
>>> mindspore.mint.tril(input)
Tensor(shape=[2, 3, 3], dtype=Int64, value=
[[[ 1,  0,  0],
  [ 5,  6,  0],
  [10, 11, 12]],
 [[ 1,  0,  0],
  [ 5,  6,  0],
  [10, 11, 12]]])
```

mindspore.mint.select

mindspore.mint.**select**(*input*, *dim*, *index*)

Slices the input tensor along the selected dimension at the given index.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – the input tensor.
- **dim** (*int*) – the dimension to slice.
- **index** (*int*) – the index to select with.

Returns

Tensor.

Raises

TypeError –If input is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> input = Tensor([[2, 3, 4, 5],[3, 2, 4, 5]])
>>> y = mint.select(input, 0, 0)
>>> print(y)
[2 3 4 5]
```

mindspore.mint.squeeze

`mindspore.mint.`**`squeeze`**(*input*, *dim*)

Return the Tensor after deleting the dimension of size 1 in the specified *dim*.

If $dim = ()$, it will remove all the dimensions of size 1. If *dim* is specified, it will remove the dimensions of size 1 in the given *dim*. For example, if the dimension is not specified $dim = ()$, input shape is (A, 1, B, C, 1, D), then the shape of the output Tensor is (A, B, C, D). If the dimension is specified, the squeeze operation is only performed in the specified dimension. If input shape is (A, 1, B), when $dim = 0$ or $dim = 2$, the input tensor is not changed, while when $dim = 1$, the input tensor shape is changed to (A, B).

Note:

- Please note that in dynamic graph mode, the output Tensor will share data with the input Tensor, and there is no Tensor data copy process.
- The dimension index starts at 0 and must be in the range *[-input.ndim, input.ndim]*.
- In GE mode, only support remove dimensions of size 1 from the shape of input tensor.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) –Used to calculate Squeeze. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **dim** (*`Union[int, tuple(int)]`*) –Specifies the dimension indexes of shape to be removed, which will remove all the dimensions of size 1 in the given dim parameter. If specified, it must be int32 or int64.

Returns

Tensor, the shape of tensor is $(x_1, x_2, ..., x_S)$.

Raises

- **TypeError** –If *input* is not a tensor.
- **TypeError** –If *dim* is not an int, tuple.

- **TypeError** –If *dim* is a tuple whose elements are not all int.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones(shape=[3, 2, 1]), mindspore.float32)
>>> output = mint.squeeze(input, 2)
>>> print(output)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

mindspore.mint.stack

mindspore.mint.**stack**(*tensors*, *dim=0*)

Stacks a list of tensors in specified dim.

Stacks the list of input tensors with the same rank *R*, output is a tensor of rank *(R+1)*.

Given input tensors of shape $(x_1, x_2, ..., x_R)$. Set the number of input tensors as *N*. If $dim \geq 0$, the shape of the output tensor is $(x_1, x_2, ..., x_{dim}, N, x_{dim+1}, ..., x_R)$.

Parameters

- **tensors** (*Union[tuple, list]*) –A Tuple or list of Tensor objects with the same shape and type.
- **dim** (*int, optional*) –Dimension to stack. The range is [-(R+1), R+1). Default: 0 .

Returns

Tensor. A stacked Tensor with the same type as *tensors*.

Raises

- **TypeError** –If the data types of elements in *tensors* are not the same.
- **ValueError** –If *dim* is out of the range [-(R+1), R+1); or if the shapes of elements in *tensors* are not the same.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> data1 = Tensor(np.array([0, 1]).astype(np.float32))
>>> data2 = Tensor(np.array([2, 3]).astype(np.float32))
>>> output = mint.stack([data1, data2], 0)
>>> print(output)
[[0. 1.]
 [2. 3.]]
```

mindspore.mint.swapaxes

mindspore.mint.**swapaxes**(*input*, *axis0*, *axis1*)

Alias for *mindspore.mint.transpose()* . The *input* corresponds to the *input* in the reference interface, and the parameters *axis0* and *axis1* correspond to *dim0* and *dim1* in the reference interface respectively.

For more details, see *mindspore.mint.transpose()* .

Warning: This is an experimental API that is subject to change or deletion.

Examples

```
>>> import numpy as np
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> input = Tensor(np.ones((2,3,4), dtype=np.float32))
>>> output = mint.swapaxes(input, 0, 2)
>>> print(output.shape)
(4, 3, 2)
```

mindspore.mint.transpose

mindspore.mint.**transpose**(*input*, *dim0*, *dim1*)

Interchange two axes of a tensor.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (*Tensor*) –Input tensor.
- **dim0** (*int*) –First axis.
- **dim1** (*int*) –Second axis.

Returns

Transposed tensor, has the same data type as *input*.

Raises

- **TypeError** –If argument *input* is not Tensor.
- **TypeError** –If *dim0* or *dim1* is not integer.
- **ValueError** –If *dim0* or *dim1* is not in the range of $[-ndim, ndim - 1]$.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones((2, 3, 4), dtype=np.float32))
>>> output = mint.transpose(input, 0, 2)
>>> print(output.shape)
(4, 3, 2)
```

mindspore.mint.triu

mindspore.mint.**triu**(*input*, *diagonal=0*)

Zero the input tensor below the diagonal specified.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **diagonal** (*int, optional*) –The diagonal specified of 2-D tensor. Default 0 represents the main diagonal.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[ 1,  2,  3,  4],
...                           [ 5,  6,  7,  8],
...                           [10, 11, 12, 13],
...                           [14, 15, 16, 17]])
>>> mindspore.mint.triu(input)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  3,  4],
 [ 0,  6,  7,  8],
 [ 0,  0, 12, 13],
 [ 0,  0,  0, 17]])
>>> mindspore.mint.triu(input, 1)
```

(continues on next page)

(continued from previous page)

```
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 0,  2,  3,  4],
 [ 0,  0,  7,  8],
 [ 0,  0,  0, 13],
 [ 0,  0,  0,  0]])
>>> mindspore.mint.triu(input, -1)
Tensor(shape=[4, 4], dtype=Int64, value=
[[ 1,  2,  3,  4],
 [ 5,  6,  7,  8],
 [ 0, 11, 12, 13],
 [ 0,  0, 16, 17]])
>>> input = mindspore.tensor([[[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]],
...                           [[ 1,  2,  3],
...                            [ 5,  6,  7],
...                            [10, 11, 12]]])
>>> mindspore.mint.triu(input)
Tensor(shape=[2, 3, 3], dtype=Int64, value=
[[[ 1,  2,  3],
  [ 0,  6,  7],
  [ 0,  0, 12]],
 [[ 1,  2,  3],
  [ 0,  6,  7],
  [ 0,  0, 12]]])
```

mindspore.mint.unbind

mindspore.mint.**unbind**(*input*, *dim=0*)

Unbind a tensor dimension in specified axis.

Given a tensor of shape $(n_1, n_2, ..., n_R)$ and unbinding it in the specified *dim*, multiple tensors with shape $(n_1, n_2, ..., n_{dim}, n_{dim+2}, ..., n_R)$ are returned.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor to unbind, with a shape of $(n_1, n_2, ..., n_R)$. The rank of the tensor must be greater than 0.
- **dim** (*int, optional*) –Dimension along which to unbind. The range is [-R, R). Default: `0` .

Returns

A tuple of tensors, the shape of each objects is the same.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not an int.
- **ValueError** –If *dim* is out of the range [-R, R).

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
>>> output = mint.unbind(input, dim=0)
>>> print(output)
(Tensor(shape=[3], dtype=Int64, value=[1, 2, 3]), Tensor(shape=[3], dtype=Int64, value=[4, 5,
 ↪ 6]),
Tensor(shape=[3], dtype=Int64, value=[7, 8, 9]))
```

mindspore.mint.unique_consecutive

mindspore.mint.**unique_consecutive**(*input*, *return_inverse=False*, *return_counts=False*, *dim=None*)

Returns the elements that are unique in each consecutive group of equivalent elements in the input tensor.

When *return_inverse=True* , it returns a tensor containing the indices of the elements in the input tensor within the output tensor.

When *return_counts=True* , it returns a tensor representing the number of occurrences of each output element in the input.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **return_inverse** (*bool, optional*) –Whether to return the index of where the element in the original input maps to the position in the output. Default: False .
- **return_counts** (*bool, optional*) –Whether to return the counts of each unique element. Default: False .
- **dim** (*int, optional*) –The dimension to apply unique. If None , the unique of the flattened input is returned. If the dimension is specified, it must be int32 or int64. Default: None .

Returns

A tensor or a tuple of tensors containing tensor objects (*output*, *inverse_indices*, *counts*).

- **output** (Tensor): the output tensor has the same type as *input* and represents the output list of unique scalar elements.
- **inverse_indices** (Tensor, optional): if *return_inverse* is True, there will be an additional returned tensor *inverse_indices*. *inverse_indices* has the same shape as *input* and represents the index of where the element in the original input maps to the position in the output.
- **counts** (Tensor, optional): if *return_counts* is True, there will be an additional returned tensor *counts*. *counts* has the same shape as *output* or *output.shape[dim]* if dim was specified and represents the number of occurrences for each unique value or tensor.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not supported.
- **TypeError** –If *return_inverse* is not a bool.

- **TypeError** – If *return_counts* is not a bool.
- **TypeError** – If *dim* is not an int.
- **ValueError** – If *dim* is not in the range of $[-ndim, ndim - 1]$.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 1, 2, 2, 3, 1, 1, 2]), mstype.int64)
>>> output, inverse_indices, counts = mint.unique_consecutive(x, True, True, None)
>>> print(output)
[1 2 3 1 2]
>>> print(inverse_indices)
[0 0 1 1 2 3 3 4]
>>> print(counts)
[2 2 1 2 1]
```

mindspore.mint.unsqueeze

mindspore.mint.**unsqueeze**(*input*, *dim*)

Adds an additional dimension to the input tensor at the given dimension.

Parameters

- **input** (Tensor) – The input tensor.
- **dim** (*int*) – The dimension specified.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, 3, 4])
>>> mindspore.mint.unsqueeze(input, 0)
Tensor(shape=[1, 4], dtype=Int64, value=
[[1, 2, 3, 4]])
>>> mindspore.mint.unsqueeze(input, 1)
Tensor(shape=[4, 1], dtype=Int64, value=
[[1],
 [2],
 [3],
 [4]])
```

mindspore.mint.where

mindspore.mint.**where**(*condition*, *input*, *other*) → *Tensor*

Selects elements from *input* or *other* based on *condition* and returns a tensor.

$$output_i = \begin{cases} input_i, & if\ condition_i \\ other_i, & otherwise \end{cases}$$

Parameters

- **condition** (*Tensor[bool]*) – If true, yield *input*, otherwise yield *other*.
- **input** (*Union[Tensor, Scalar]*) – When *condition* is true, values to select from.
- **other** (*Union[Tensor, Scalar]*) – When *condition* is false, values to select from.

Returns

Tensor, elements are selected from *input* and *other*.

Raises

- **TypeError** – If *condition* is not a tensor.
- **TypeError** – If both *input* and *other* are scalars.
- **ValueError** – If *condition*, *input* and *other* can not broadcast to each other.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import tensor, mint
>>> from mindspore import dtype as mstype
>>> a = tensor(np.arange(4).reshape((2, 2)), mstype.float32)
>>> b = tensor(np.ones((2, 2)), mstype.float32)
>>> condition = a < 3
>>> output = mint.where(condition, a, b)
>>> print(output)
[[0. 1.]
 [2. 1.]]
```

mindspore.mint.**where**(*condition*) → *Tensor*

Identical to *mindspore.ops.nonzero()* with input *condition* and *as_tuple* being True.

Supported Platforms:

Ascend

4.2 Random Sampling

API Name	Description	Supported Platforms
mindspore.mint.multinomial	Returns a tensor sampled from the multinomial probability distribution located in the corresponding row of the input tensor.	Ascend
mindspore.mint.normal	Generates random numbers according to the standard Normal (or Gaussian) random number distribution.	Ascend
mindspore.mint.rand_like	Returns a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given dtype and shape of the input tensor.	Ascend
mindspore.mint.rand	Returns a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given shape and dtype.	Ascend

4.2.1 mindspore.mint.multinomial

mindspore.mint.**multinomial**(*input*, *num_samples*, *replacement=False*, ***, *generator=None*)

Returns a tensor sampled from the multinomial probability distribution located in the corresponding row of the input tensor.

The polynomial distribution is a probability distribution that generalizes the binomial distribution formula to multiple states. In the polynomial distribution, each event has a fixed probability, and the sum of these probabilities is 1. The purpose of the *mindspore.mint.multinomial()* interface is to perform *num_samples* sampling on the input *input*, and the output tensor is the index of the input tensor for each sampling. The values in *input* represent the probability of selecting the corresponding index for each sampling.

Here is an extreme example for better understanding. Suppose we have an input probability tensor with values *Tensor([90 / 100, 10 / 100, 0], mindspore.float32)*, which means we can sample three indices, namely index 0, index 1, and index 2, with probabilities of 90%, 10%, and 0%, respectively. We perform n samplings, and the resulting sequence is the calculation result of the polynomial distribution, with a length equal to the number of samplings.

In case 1 of the sample code, we perform two non-replacement samplings (*replacement* is *False*). The calculation result is most likely *[0, 1]*, and less likely *[1, 0]*. Since the probability of selecting index 0 is 90% for each sampling, the first result is most likely to be index 0. Since the probability of selecting index 2 is 0, index 2 cannot appear in the sampling result. Therefore, the second result must be index 1, and the resulting sequence is *[0, 1]*.

In case 2 of the sample code, we perform 10 replacement samplings (*replacement* is *True*). As expected, about 90% of the sampling results are index 0.

In case 3 of the sample code, we extend the input to 2 dimensions, and the sampling results in each dimension also match our sampling expectations.

Note: The rows of input do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum. When using values as weights, it can be understood as normalizing the input along the last dimension.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor containing probabilities, must be 1 or 2 dimensions, with float32 data type.

- **num_samples** (*int*) –Number of samples to draw.
- **replacement** (*bool, optional*) –Whether to draw with replacement or not. Default: `False`.

Keyword Arguments

generator (*generator, optional*) –MindSpore generator. Default: `None`.

Returns

Tensor, dtype is Int64. If *input* is a vector, out is a vector of size *num_samples*. If *input* is a matrix with m rows, out is an matrix of shape(m * num_samples).

Raises

- **TypeError** –If *input* is not a Tensor whose dtype is not in float16, float32, float64 or bfloat16.
- **TypeError** –If *num_samples* is not an int, a Scalar of int or a Tensor with shape[1,] and only one int element.
- **RuntimeError** –If num_samples <= 0.
- **RuntimeError** –If *replacement* is False, num_samples > *shape* of the last dimension of *input*.
- **RuntimeError** –If shape of the last dimension of *input* exceeds `2^24`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> # case 1: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is False.
>>> input1 = Tensor([90 / 100, 10 / 100, 0], mindspore.float32)
>>> input2 = Tensor([90, 10, 0], mindspore.float32)
>>> # input1 and input2 have the same meaning.
>>> output1 = mint.multinomial(input1, 2)
>>> output2 = mint.multinomial(input2, 2)
>>> # print(output1)
>>> # [0 1]
>>> # print(output2)
>>> # [0 1]
>>> print(len(output1))
2
>>> print(len(output2))
2
>>> # case 2: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is True.
>>> output3 = mint.multinomial(input1, 10, replacement=True)
>>> # print(output3)
>>> # [0 0 1 0 0 0 0 0 0 0]
>>> print(len(output3))
10
>>> # case 3: The output is random, and the length of the output is the same as num_sample.
>>> # replacement is True.
>>> # rank is 2
>>> input4 = Tensor([[90, 10, 0], [10, 90, 0]], mstype.float32)
```

(continues on next page)

(continued from previous page)

```
>>> output4 = mint.multinomial(input4, 10, replacement=True)
>>> # print(output4)
>>> # [[0 0 0 0 0 0 0 0 1 0]
>>> #  [1 1 1 1 1 0 1 1 1 1]]
```

4.2.2 mindspore.mint.normal

mindspore.mint.**normal**(*mean*, *std*, *, *generator=None*) → *Tensor*

Generates random numbers according to the standard Normal (or Gaussian) random number distribution.

Parameters

- **mean** (*Union[float, Tensor]*) –Mean value of each element, the shape of the *mean* tensor should be the same as that of the *std* tensor.
- **std** (*Union[float, Tensor]*) –Standard deviation for each element, the shape of the *std* tensor should be the same as that of the *mean* tensor. The value of *std* should be greater than or equal to 0.

Keyword Arguments

generator (*generator, optional*) –MindSpore generator. Default: `None`.

Returns

Outputs a tensor with the same shape as *mean*.

Raises

TypeError –If *mean* or *std* is not Union[float, Tensor].

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> mean = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> std = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> output = mint.normal(mean, std)
>>> print(output.shape)
(3,)
```

mindspore.mint.**normal**(*mean*, *std=1.0*) → *Tensor*

Similar to the function above, but the standard deviations are shared among all drawn elements.

Parameters

- **mean** (`Tensor`) –Mean value of each element.
- **std** (*float, optional*) –Standard deviation for each element. The value of *std* should be greater than or equal to 0. Default: `1.0`.

Returns

Outputs a tensor with the same shape as *mean*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> mean = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> output = mint.normal(mean, 1.0)
>>> print(output.shape)
(3,)
```

`mindspore.mint.`**`normal`**(*mean*, *std*, *size*) → *Tensor*

Similar to the function above, but the means and standard deviations are shared among all drawn elements. The result tensor has size given by *size*.

Parameters

- **`mean`** (*float*) –Mean value of each element.
- **`std`** (*float*) –Standard deviation for each element.
- **`size`** (*tuple*) –output shape.

Returns

Outputs a tensor. The shape is specified as *size*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> output = mint.normal(1.0, 2.0, (2, 4))
>>> print(output.shape)
(2, 4)
```

4.2.3 mindspore.mint.rand_like

mindspore.mint.**rand_like**(*input*, *, *dtype=None*)

Returns a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given dtype and shape of the input tensor.

Parameters

input (Tensor) −Input Tensor to specify the output shape and its default dtype.

Keyword Arguments

dtype (*mindspore.dtype*, optional) −Designated tensor dtype, it must be float type. If None, the same dtype of *input* will be applied. Default: None .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the uniform distribution on the interval $[0, 1)$.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import Tensor, mint
>>> a = Tensor([[2, 3, 4], [1, 2, 3]])
>>> print(mint.rand_like(a, dtype=ms.float32).shape)
(2, 3)
```

4.2.4 mindspore.mint.rand

mindspore.mint.**rand**(**size*, *generator=None*, *dtype=None*)

Returns a new tensor that fills numbers from the uniform distribution over an interval $[0, 1)$ based on the given shape and dtype.

Parameters

size (*Union[int, tuple(int), list(int)]*) −Shape of the new tensor, e.g. $(2, 3)$ or 2.

Keyword Arguments

- **generator** (*mindspore.Generator*, optional) −a pseudorandom number generator. Default: None, uses the default pseudorandom number generator.
- **dtype** (*mindspore.dtype*, optional) −Designated tensor dtype. If None, *mindspore.float32* will be applied. Default: None .

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the uniform distribution on the interval $[0, 1)$.

Raises

ValueError −If *size* contains negative numbers.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint
>>> print(mint.rand(2, 3).shape)
(2, 3)
```

4.3 Math Operations

4.3.1 Pointwise Operations

API Name	Description	Supported Platforms
mindspore.mint.abs	Compute the absolute value of a tensor element-wise.	Ascend
mindspore.mint.add	Adds scaled other value to *self*.	Ascend
mindspore.mint.addmv	Performs a matrix-vector product of *mat* and *vec*, and add the input vector *input* to the final result.	Ascend
mindspore.mint.acos	Computes arccosine of input tensors element-wise.	Ascend
mindspore.mint.acosh	Computes inverse hyperbolic cosine of the inputs element-wise.	Ascend
mindspore.mint.arccos	Alias for mindspore.mint.acos() .	Ascend
mindspore.mint.arccosh	Alias for mindspore.mint.acosh().	Ascend
mindspore.mint.arcsin	Alias for mindspore.mint.asin().	Ascend
mindspore.mint.arcsinh	Alias for mindspore.mint.asinh().	Ascend
mindspore.mint.arctan	Alias for mindspore.mint.atan().	Ascend
mindspore.mint.arctan2	Alias for mindspore.mint.atan2().	Ascend
mindspore.mint.arctanh	Alias for mindspore.mint.atanh().	Ascend
mindspore.mint.asin	Computes arcsine of input tensors element-wise.	Ascend
mindspore.mint.asinh	Computes inverse hyperbolic sine of the input element-wise.	Ascend
mindspore.mint.atan	Computes the trigonometric inverse tangent of the input element-wise.	Ascend
mindspore.mint.atan2	Returns arctangent of input/other element-wise.	Ascend
mindspore.mint.atanh	Computes inverse hyperbolic tangent of the input element-wise.	Ascend
mindspore.mint.bitwise_and	Returns bitwise *and* of two tensors element-wise.	Ascend
mindspore.mint.bitwise_or	Returns bitwise *or* of two tensors element-wise.	Ascend
mindspore.mint.bitwise_xor	Returns bitwise *xor* of two tensors element-wise.	Ascend
mindspore.mint.ceil	Rounds a tensor up to the closest integer element-wise.	Ascend
mindspore.mint.clamp	Clamps tensor values between the specified minimum value and maximum value.	Ascend
mindspore.mint.cos	Computes cosine of input element-wise.	Ascend
mindspore.mint.cosh	Computes hyperbolic cosine of input element-wise.	Ascend
mindspore.mint.cross	Compute the cross product of two input tensors along the specified dimension.	Ascend
mindspore.mint.diff	Computes the n-th forward difference along the given dimension.	Ascend
mindspore.mint.div	Divides each element of the *input* by the corresponding element of the *other* .	Ascend

continues on next page

Table 4 – continued from previous page

mindspore.mint.divide	Alias for mindspore.mint.div() .	Ascend
mindspore.mint.erf	Compute the Gauss error of input tensor element-wise.	Ascend
mindspore.mint.erfc	Compute the complementary error function of input tensor element-wise.	Ascend
mindspore.mint.erfinv	Compute the inverse error of input tensor element-wise.	Ascend
mindspore.mint.exp	Compute exponential of the input tensor element-wise.	Ascend
mindspore.mint.exp2	Calculates the base-2 exponent of the Tensor *input* element by element.	Ascend
mindspore.mint.expm1	Compute exponential of the input tensor, then minus 1, element-wise.	Ascend
mindspore.mint.fix	Alias for mindspore.mint.trunc() .	Ascend
mindspore.mint.float_power	Computes *input* to the power of *exponent* element-wise in double precision, and always returns a mindspore.float64 tensor.	Ascend
mindspore.mint.floor	Rounds a tensor down to the closest integer element-wise.	Ascend
mindspore.mint.fmod	Computes the floating-point remainder of the division operation input/other.	Ascend
mindspore.mint.frac	Calculates the fractional part of each element in the input.	Ascend
mindspore.mint.lerp	Perform a linear interpolation of two tensors input and end based on a float or tensor weight.	Ascend
mindspore.mint.log	Compute the natural logarithm of the input tensor element-wise.	Ascend
mindspore.mint.log1p	Compute the natural logarithm of (tensor + 1) element-wise.	Ascend
mindspore.mint.log2	Returns the logarithm to the base 2 of a tensor element-wise.	Ascend
mindspore.mint.log10	Returns the logarithm to the base 10 of a tensor element-wise.	Ascend
mindspore.mint.logaddexp	Computes the logarithm of the sum of exponentiations of the inputs.	Ascend
mindspore.mint.logaddexp2	Logarithm of the sum of exponentiations of the inputs in base of 2.	Ascend
mindspore.mint.logical_and	Compute the "logical AND" of two tensors element-wise.	Ascend
mindspore.mint.logical_not	Compute the "logical NOT" of the input tensor element-wise.	Ascend
mindspore.mint.logical_or	Compute the "logical OR" of two tensors element-wise.	Ascend
mindspore.mint.logical_xor	Compute the "logical XOR" of two tensors element-wise.	Ascend
mindspore.mint.mul	Multiply other value by input Tensor.	Ascend
mindspore.mint.mv	Multiply matrix *input* and vector *vec*.	Ascend
mindspore.mint.nansum	Computes sum of *input* over a given dimension, treating NaNs as zero.	Ascend

continues on next page

Table 4 – continued from previous page

mindspore.mint.nan_to_num	Replace the *NaN*, positive infinity and negative infinity values in *input* with the specified values in *nan*, *posinf* and *neginf* respectively.	Ascend
mindspore.mint.neg	Returns a tensor with negative values of the input tensor element-wise.	Ascend
mindspore.mint.negative	Alias for mindspore.mint.neg() .	Ascend
mindspore.mint.pow	Calculates the *exponent* power of each element in *input*.	Ascend
mindspore.mint.polar	Converts polar coordinates to Cartesian coordinates.	Ascend
mindspore.mint.ravel	Expand the multidimensional Tensor into 1D along the 0 axis direction.	Ascend
mindspore.mint.reciprocal	Returns reciprocal of a tensor element-wise.	Ascend
mindspore.mint.remainder	Computes the remainder of *input* divided by *other* element-wise.	Ascend
mindspore.mint.roll	Roll the elements of a tensor along a dimension.	Ascend
mindspore.mint.round	Round elements of input to the nearest integer.	Ascend
mindspore.mint.rsqrt	Compute reciprocal of square root of input tensor element-wise.	Ascend
mindspore.mint.sigmoid	Computes Sigmoid of input element-wise.	Ascend
mindspore.mint.sign	Return an element-wise indication of the sign of a number.	Ascend
mindspore.mint.sin	Compute sine of the input tensor element-wise.	Ascend
mindspore.mint.sinc	Compute the normalized sinc of input.	Ascend
mindspore.mint.sinh	Compute hyperbolic sine of the input element-wise.	Ascend
mindspore.mint.softmax	Alias for mindspore.mint.nn.functional.softmax().	Ascend
mindspore.mint.sqrt	Returns sqrt of a tensor element-wise.	Ascend
mindspore.mint.square	Return square of a tensor element-wise.	Ascend
mindspore.mint.sub	Subtracts scaled other value from self Tensor.	Ascend
mindspore.mint.t	Transpose the input tensor.	Ascend
mindspore.mint.tan	Computes tangent of *input* element-wise.	Ascend
mindspore.mint.tanh	Computes hyperbolic tangent of input element-wise.	Ascend
mindspore.mint.trunc	Returns a tensor with the truncated integer values of the elements of the input tensor.	Ascend
mindspore.mint.xlogy	Computes the first input multiplied by the logarithm of second input element-wise.	Ascend

mindspore.mint.abs

mindspore.mint.**abs**(*input*)

Compute the absolute value of a tensor element-wise.

$$out_i = |input_i|$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.abs(mindspore.tensor([-1.0, 1.0, 0.0]))
>>> print(output)
[1. 1. 0.]
```

mindspore.mint.add

mindspore.mint.**add**(*input*, *other*, *, *alpha=1*) → *Tensor*

Adds scaled other value to *self*.

$$out_i = self_i + alpha \times other_i$$

Note:

- When *self* and *other* have different shapes, they must be able to broadcast to a common shape.
- *self*, *other* and *alpha* comply with the implicit type conversion rules to make the data types consistent.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – *input* is a number.Number or a bool or a tensor whose data type is number or bool_.
- **other** (*Union[Tensor, number.Number, bool]*) – *other* is a number.Number or a bool or a tensor whose data type is number or bool_.

Keyword Arguments

alpha (*number.Number, optional*) – A scaling factor applied to *other*, default 1.

Returns

Tensor with a shape that is the same as the broadcasted shape of the *self* and *other*, and the data type is the one with higher precision or higher digits among *self*, *other* and *alpha*.

Raises

- **TypeError** – If the type of *other* or *alpha* is not one of the following: Tensor, number.Number, bool.
- **TypeError** – If *alpha* is of type float but *self* and *other* are not of type float.
- **TypeError** – If *alpha* is of type bool but *self* and *other* are not of type bool.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> x = Tensor(1, mindspore.int32)
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> alpha = 0.5
>>> output = mint.add(x, y, alpha=alpha)  # x.add(y, alpha=alpha)
>>> print(output)
[3. 3.5 4.]
>>> # the data type of x is int32, the data type of y is float32,
>>> # alpha is a float, and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

mindspore.mint.addmv

mindspore.mint.**addmv**(*input*, *mat*, *vec*, *, *beta=1*, *alpha=1*)

Performs a matrix-vector product of *mat* and *vec*, and add the input vector *input* to the final result.

If *mat* is a tensor of size (N, M) , *vec* is a 1-D tensor of size M , then *input* must be broadcastable with a 1-D tensor of size N . In this case, *output* is a 1-D Tensor of size N .

$$output = \beta input + \alpha(mat@vec)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Vector to be added.
- **mat** (Tensor) –The first tensor needs to be multiplied.
- **vec** (Tensor) –The second tensor needs to be multiplied.

Keyword Arguments

- **beta** (*Union[float, int], optional*) –Coefficient of *input*. Default: 1.
- **alpha** (*Union[float, int], optional*) –Coefficient of $mat@vec$. Default: 1.

Returns

Tensor, with a shape of $(N,)$, and its dtype is the same as *input*.

Raises

- **TypeError** –If dtype of *input*, *mat* or *vec* is not tensor.
- **TypeError** –If dtypes of *mat* and *vec* are not the same.
- **ValueError** –If *mat* is not a 2-D tensor.
- **ValueError** –If *vec* is not a 1-D tensor.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([2., 3.]).astype(np.float32))
>>> mat = Tensor(np.array([[2., 5., 3.], [4., 2., 2.]]).astype(np.float32))
>>> vec = Tensor(np.array([3., 2., 4.]).astype(np.float32))
>>> output = mint.addmv(input, mat, vec)
>>> print(output)
[30. 27.]
```

mindspore.mint.acos

mindspore.mint.**acos**(*input*)

Computes arccosine of input tensors element-wise.

$$out_i = \cos^{-1}(input_i)$$

Parameters

input (Tensor) −The shape of tensor is $(N, *)$, where $*$ means any number of additional dimensions.

Returns

Tensor, has the same shape as *input*. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

TypeError −If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0.74, 0.04, 0.30, 0.56]), mindspore.float32)
>>> output = mint.acos(input)
>>> print(output)
[0.7377037  1.5307857 1.2661037 0.9764114]
```

mindspore.mint.acosh

mindspore.mint.**acosh**(*input*)

Computes inverse hyperbolic cosine of the inputs element-wise.

$$out_i = \cosh^{-1}(input_i)$$

Note: Given an input tensor input, the function computes inverse hyperbolic cosine of every element. Input range is [1, inf].

Parameters

input (Tensor) – The input tensor of inverse hyperbolic cosine function.

Returns

Tensor, has the same shape as *input*. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1.0, 1.5, 3.0, 100.0]), mindspore.float32)
>>> output = mint.acosh(input)
>>> print(output)
[0.        0.9624236 1.7627472 5.298292 ]
```

mindspore.mint.arccos

mindspore.mint.**arccos**(*input*)

Alias for *mindspore.mint.acos()*.

Supported Platforms:

Ascend

mindspore.mint.arccosh

mindspore.mint.**arccosh**(*input*)

Alias for *mindspore.mint.acosh()*.

Supported Platforms:

Ascend

mindspore.mint.arcsin

mindspore.mint.**arcsin**(*x*)

Alias for *mindspore.mint.asin()*.

Supported Platforms:

Ascend

mindspore.mint.arcsinh

mindspore.mint.**arcsinh**(*input*)

Alias for *mindspore.mint.asinh()*.

Supported Platforms:

Ascend

mindspore.mint.arctan

mindspore.mint.**arctan**(*input*)

Alias for *mindspore.mint.atan()*.

Supported Platforms:

Ascend

mindspore.mint.arctan2

mindspore.mint.**arctan2**(*input*, *other*)

Alias for *mindspore.mint.atan2()*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.array([0, 1]), mindspore.float32)
>>> y = Tensor(np.array([1, 1]), mindspore.float32)
>>> output = mint.arctan2(x, y)
>>> print(output)
[0.        0.7853982]
```

mindspore.mint.arctanh

mindspore.mint.**arctanh**(*input*)

Alias for *mindspore.mint.atanh()*.

Supported Platforms:

Ascend

mindspore.mint.asin

mindspore.mint.**asin**(*input*)

Computes arcsine of input tensors element-wise.

$$out_i = \sin^{-1}(input_i)$$

Parameters

input (Tensor) –The shape of tensor is $(N, *)$, where $*$ means any number of additional dimensions.

Returns

Tensor, has the same shape as *input*. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0.74, 0.04, 0.30, 0.56]), mindspore.float32)
>>> output = mint.asin(input)
>>> print(output)
[0.8330927  0.04001068  0.30469266  0.59438497 ]
```

mindspore.mint.asinh

mindspore.mint.**asinh**(*input*)

Computes inverse hyperbolic sine of the input element-wise.

$$out_i = \sinh^{-1}(input_i)$$

Parameters

input (Tensor) –The input tensor of inverse hyperbolic sine function.

Returns

Tensor, has the same shape as *input*. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-5.0, 1.5, 3.0, 100.0]), mindspore.float32)
>>> output = mint.asinh(input)
>>> print(output)
[-2.3124385  1.1947632  1.8184465  5.298342 ]
```

mindspore.mint.atan

mindspore.mint.**atan**(*input*)

Computes the trigonometric inverse tangent of the input element-wise.

$$out_i = \tan^{-1}(input_i)$$

Parameters

input (Tensor) – The shape of tensor is $(N, *)$ where $*$ means, any number of additional dimensions.

Returns

Tensor, has the same shape as *input*. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1.0, 0.0]), mindspore.float32)
>>> output = mint.atan(input)
>>> print(output)
[0.7853982 0.       ]
```

mindspore.mint.atan2

mindspore.mint.**atan2**(*input*, *other*)

Returns arctangent of input/other element-wise.

It returns $\theta \in [-\pi, \pi]$ such that $input = r * \sin(\theta), other = r * \cos(\theta)$, where $r = \sqrt{input^2 + other^2}$.

Note:

- Arg *input* and *other* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to relatively the highest precision data type.

Parameters

- **input** (Tensor, *Number.number*) –The input tensor or scalar.
- **other** (Tensor, *Number.number*) –The input tensor or scalar. It has the same shape with *input* or its shape is able to broadcast with *input*.

Returns

Tensor, the shape is the same as the one after broadcasting. The dtype of output is float32 when dtype of *input* is in [bool, int8, uint8, int16, int32, int64]. Otherwise output has the same dtype as *input*.

Raises

- **TypeError** –If *input* or *other* is not a Tensor or scalar.
- **RuntimeError** –If the data type of *input* and *other* conversion of Parameter is required when data type conversion of Parameter is not supported.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0, 1]), mindspore.float32)
>>> other = Tensor(np.array([1, 1]), mindspore.float32)
>>> output = mint.atan2(input, other)
>>> print(output)
[0.        0.7853982]
```

mindspore.mint.atanh

mindspore.mint.**atanh**(*input*)

Computes inverse hyperbolic tangent of the input element-wise.

$$out_i = \tanh^{-1}(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.atanh(mindspore.tensor([0, -0.5]))
>>> print(output)
[ 0.         -0.54930615]
```

mindspore.mint.bitwise_and

mindspore.mint.**bitwise_and**(*input*, *other*)

Returns bitwise *and* of two tensors element-wise.

$$out_i = input_i \wedge other_i$$

Note: Args of *input* and *other* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to the relatively highest precision data type.

Parameters

- **input** (Tensor) – The input tensor.
- **other** (Tensor, *Number.number*) – The input tensor or scalar. It has the same shape with *input* or its shape is able to broadcast with *input*.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is same as *input*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = mint.bitwise_and(input, other)
>>> print(output)
[ 0  0  1 -1  1  0  1]
```

mindspore.mint.bitwise_or

mindspore.mint.**bitwise_or**(*input*, *other*)

Returns bitwise *or* of two tensors element-wise.

$$out_i = input_i \mid other_i$$

Note: Args of *input* and *other* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to the relatively highest precision data type.

Parameters

- **input** (Tensor) – The input tensor.
- **other** (Tensor, *Number.number*) – The input tensor or scalar. It has the same shape with *input* or its shape is able to broadcast with *input*.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is same as *input*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = mint.bitwise_or(input, other)
>>> print(output)
[ 0  1  1 -1 -1  3  3]
```

mindspore.mint.bitwise_xor

mindspore.mint.**bitwise_xor**(*input*, *other*)

Returns bitwise *xor* of two tensors element-wise.

$$out_i = input_i \oplus other_i$$

Note: Args of *input* and *other* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to the relatively highest precision data type.

Parameters

- **input** (Tensor) – The input tensor.

- **other** (Tensor, *Number.number*) –The input tensor or scalar. It has the same shape with *input* or its shape is able to broadcast with *input*.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is same as *input*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> other = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> output = mint.bitwise_xor(input, other)
>>> print(output)
[ 0  1  0  0 -2  3  2]
```

mindspore.mint.ceil

mindspore.mint.**ceil**(*input*)

Rounds a tensor up to the closest integer element-wise.

$$out_i = \lceil input_i \rceil$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.1, 2.5, -1.5])
>>> output = mindspore.mint.ceil(input)
>>> print(output)
[ 2.  3. -1.]
```

mindspore.mint.clamp

mindspore.mint.**clamp**(*input*, *min=None*, *max=None*) → *Tensor*

Clamps tensor values between the specified minimum value and maximum value.

Limits the value of $input$ to a range, whose lower limit is *min* and upper limit is *max* .

$$out_i = \begin{cases} max & \text{if } input_i \geq max \\ input_i & \text{if } min < input_i < max \\ min & \text{if } input_i \leq min \end{cases}$$

Note:

- *min* and *max* cannot be None at the same time;
- When *min* is None and *max* is not None, the elements in Tensor larger than *max* will become *max*;
- When *min* is not None and *max* is None, the elements in Tensor smaller than *min* will become *min*;
- If *min* is greater than *max*, the value of all elements in Tensor will be set to *max*;
- The data type of *input*, *min* and *max* should support implicit type conversion and cannot be bool type.

Parameters

- **input** (Tensor) –Input data, which type is Tensor. Tensors of arbitrary dimensions are supported.
- **min** (*Union(Tensor, float, int), optional*) –The minimum value. Default: None .
- **max** (*Union(Tensor, float, int), optional*) –The maximum value. Default: None .

Returns

Tensor, a clipped Tensor. The data type and shape are the same as input.

Raises

- **ValueError** –If both *min* and *max* are None.
- **TypeError** –If the type of *input* is not Tensor.
- **TypeError** –If the type of *min* is not in None, Tensor, float or int.
- **TypeError** –If the type of *max* is not in None, Tensor, float or int.

Supported Platforms:

Ascend

Examples

```
>>> # case 1: the data type of input is Tensor
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> min_value = Tensor(5, mindspore.float32)
>>> max_value = Tensor(20, mindspore.float32)
>>> input = Tensor(np.array([[1., 25., 5., 7.], [4., 11., 6., 21.]]), mindspore.float32)
>>> output = mint.clamp(input, min_value, max_value)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[ 5. 20.  5.  7.]
 [ 5. 11.  6. 20.]]
>>> # case 2: the data type of input is number
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> min_value = 5
>>> max_value = 20
>>> input = Tensor(np.array([[1., 25., 5., 7.], [4., 11., 6., 21.]]), mindspore.float32)
>>> output = mint.clamp(input, min_value, max_value)
>>> print(output)
[[ 5. 20.  5.  7.]
 [ 5. 11.  6. 20.]]
```

mindspore.mint.cos

mindspore.mint.**cos**(*input*)

Computes cosine of input element-wise.

$$out_i = \cos(x_i)$$

Warning: Using float64 may cause a problem of missing precision.

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [bool, int8, uint8,
 ↪ int16, int32, int64](Datatype only supported on Ascend).
>>> input = mindspore.tensor([0, 1, 2], mindspore.int32)
>>> mindspore.mint.cos(input)
Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  5.40302306e-01, -4.16146837e-01])
>>>
>>> # Otherwise output has the same dtype as the `input`.
>>> input = mindspore.tensor([0.74, 0.04, 0.30, 0.56], mindspore.float64)
>>> mindspore.mint.cos(input)
Tensor(shape=[4], dtype=Float64, value= [ 7.38468559e-01,  9.99200107e-01,  9.55336489e-01, ␣
 ↪8.47255111e-01])
```

mindspore.mint.cosh

mindspore.mint.**cosh**(*input*)

Computes hyperbolic cosine of input element-wise.

$$out_i = \cosh(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.74, 0.04, 0.30, 0.56])
>>> output = mindspore.mint.cosh(input)
>>> print(output)
[1.2865248 1.0008001 1.0453385 1.1609408]
```

mindspore.mint.cross

mindspore.mint.**cross**(*input*, *other*, *dim=None*)

Compute the cross product of two input tensors along the specified dimension.

Note: *input* and *other* must have the same shape, and the size of their *dim* dimension should be *3*. If *dim* is not specified, it is set to be the first dimension found with the size *3*.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.
- **dim** (*int, optional*) –Specify the dimension for computation. Default None.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
>>> other = mindspore.tensor([[4, 5, 6], [4, 5, 6], [4, 5, 6]])
>>> mindspore.mint.cross(input, other)
Tensor(shape=[3, 3], dtype=Int64, value=
[[0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]])
>>> mindspore.mint.cross(input, other, dim=1)
Tensor(shape=[3, 3], dtype=Int64, value=
[[-3,  6, -3],
 [-3,  6, -3],
 [-3,  6, -3]])
```

mindspore.mint.diff

mindspore.mint.**diff**(*input*, *n=1*, *dim=- 1*, *prepend=None*, *append=None*)

Computes the n-th forward difference along the given dimension.

The first-order differences are given by $out[i] = input[i+1] - input[i]$. Higher-order differences are calculated by using *diff* recursively.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – the tensor to compute the differences on.
- **n** (*int, optional*) – the number of times to recursively compute the difference. Default: `1` .
- **dim** (*int, optional*) – the dimension to compute the difference along. Default is the last dimension. Default: `0` .
- **prepend** (Tensor, *optional*) – values to prepend or append to *input* along *dim* before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input's shape except on *dim*. Default: `None` .
- **append** (Tensor, *optional*) – values to prepend or append to *input* along *dim* before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input's shape except on *dim*. Default: `None` .

Returns

Tensor, the result of n-th forward difference computation.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If *n* is not a scalar or scalar tensor.
- **TypeError** – If *dim* is not a scalar or scalar tensor.
- **TypeError** – If *input* type is complex64, complex128, float64, int16.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> x = Tensor([1, 3, -1, 0, 4])
>>> out = mint.diff(x)
>>> print(out.asnumpy())
[ 2 -4  1  4]
```

mindspore.mint.div

mindspore.mint.**div**(*input*, *other*, *, *rounding_mode=None*) → *Tensor*

Divides each element of the *input* by the corresponding element of the *other* .

$$out_i = input_i / other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- The two inputs comply with the implicit type conversion rules to make the data types consistent.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The dividend.
- **other** (*Union[Tensor, Number, bool]*) – The divisor.

Keyword Arguments

rounding_mode (*str, optional*) – Type of rounding applied to the result. Default: None . Three types are defined as,

- None: Default behavior, which is the same as true division in Python or *true_divide* in NumPy.
- "floor": Rounds the division of the inputs down, which is the same as floor division in Python or *floor_divide* in NumPy.
- "trunc": Rounds the division of the inputs towards zero, which is the same as C-style integer division.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** – If *input* and *other* is not one of the following: Tensor, Number, bool.
- **ValueError** – If *rounding_mode* value is not None, "floor" or "trunc".

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([4.0, 5.0, 6.0]), mindspore.float32)
>>> output = mint.div(x, y)
>>> print(output)
[0.25 0.4 0.5]
```

mindspore.mint.divide

mindspore.mint.**divide**(*input*, *other*, *, *rounding_mode=None*)

Alias for *mindspore.mint.div()* .

Supported Platforms:

Ascend

mindspore.mint.erf

mindspore.mint.**erf**(*input*)

Compute the Gauss error of input tensor element-wise.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [int64,␣
↪bool](Datatype only supported on Ascend).
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.int64)
>>> mindspore.mint.erf(input)
Tensor(shape=[5], dtype=Float32, value= [-8.42700793e-01,  0.00000000e+00,  8.42700793e-01, ␣
↪9.95322265e-01,  9.99977910e-01])
>>>
>>> # Otherwise output has the same dtype as the input.
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.float64)
>>> mindspore.mint.erf(input)
Tensor(shape=[5], dtype=Float64, value= [-8.42700793e-01,  0.00000000e+00,  8.42700793e-01, ␣
↪9.95322265e-01,  9.99977910e-01])
```

mindspore.mint.erfc

mindspore.mint.**erfc**(*input*)

Compute the complementary error function of input tensor element-wise.

$$\operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [int64,␣
↪bool](Datatype only supported on Ascend).
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.int64)
>>> mindspore.mint.erfc(input)
Tensor(shape=[5], dtype=Float32, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
>>>
>>> # Otherwise output has the same dtype as the input.
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.float64)
>>> mindspore.mint.erfc(input)
Tensor(shape=[5], dtype=Float64, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
```

mindspore.mint.erfinv

mindspore.mint.**erfinv**(*input*)

Compute the inverse error of input tensor element-wise.

It is defined in the range *(-1, 1)* as:

$$erfinv(erf(x)) = x$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # When the `input` is int8, int16, int32, int64, uint8 or bool, the return value type is␣
↪float32.
>>> input = mindspore.tensor([0, 0.5, -0.9], mindspore.int64)
>>> mindspore.mint.erfinv(input)
Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00])
>>> # Otherwise, the return value type is the same as the input type.
>>> input = mindspore.tensor([0, 0.5, -0.9], mindspore.float32)
>>> mindspore.mint.erfinv(input)
Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  4.76936132e-01, -1.16308689e+00])
```

mindspore.mint.exp

mindspore.mint.**exp**(*input*)

Compute exponential of the input tensor element-wise.

$$out_i = e^{x_i}$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.mint.exp(input)
>>> print(output)
[ 1.        2.7182817 20.085537]
```

mindspore.mint.exp2

mindspore.mint.**exp2**(*input*)

Calculates the base-2 exponent of the Tensor *input* element by element.

$$out_i = 2^{input_i}$$

Parameters

input (Tensor) −The input Tensor.

Returns

Tensor, which has the same shape as the *input*.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 1.0, 2.0, 4.0]), mindspore.float32)
>>> output = mint.exp2(x)
>>> print(output)
[ 1. 2. 4. 16.]
```

mindspore.mint.expm1

mindspore.mint.**expm1**(*input*)

Compute exponential of the input tensor, then minus 1, element-wise.

$$out_i = e^{x_i} - 1$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.mint.expm1(input)
>>> print(output)
[ 0.         1.7182817 19.085537 ]
```

mindspore.mint.fix

mindspore.mint.**fix**(*input*)

Alias for *mindspore.mint.trunc()*.

For more details, see *mindspore.mint.trunc()*.

Supported Platforms:

Ascend

mindspore.mint.float_power

mindspore.mint.**float_power**(*input*, *exponent*)

Computes *input* to the power of *exponent* element-wise in double precision, and always returns a mindspore.float64 tensor.

$$out_i = input_i^{exponent_i}$$

Warning: This is an experimental API that is subject to change or deletion.

Note: Unlike *ops.pow*, this function always uses double precision for calculations, while the precision of *ops.pow* depends on type promotion. Currently, this function does not support complex number calculations. Since float64 calculations are significantly slower on ascend devices compared to other data types, it is strongly recommended to use this function only in scenarios where double precision is required and performance is not a priority. Otherwise, using *ops.pow* is a better choice.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input is a tensor or a number.
- **exponent** (*Union[Tensor, Number]*) – The second input, if the first input is Tensor, the second input can be Number or Tensor. Otherwise, it must be a Tensor.

Returns

Tensor, the shape is the same as the one after broadcasting, the return value type is mindspore.float64.

Raises

- **TypeError** – If neither *input* nor *exponent* is a Tensor.
- **TypeError** – If the data type of *input* or *exponent* is neither a tensor nor a number, or it contains complex numbers.
- **ValueError** – If *input* and *exponent* have different shapes and cannot be broadcasted to each other.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor, mint
>>> input = Tensor([1, 2, 3])
>>> mint.float_power(input, 2)
Tensor(shape=[3], dtype=Float64, value= [ 1.00000000e+00,  4.00000000e+00,  9.00000000e+00])
>>>
>>> exp = Tensor([2, -3, -4])
>>> mint.float_power(input, exp)
Tensor(shape=[3], dtype=Float64, value= [ 1.00000000e+00,  1.25000000e-01,  1.23456790e-02])
```

mindspore.mint.floor

mindspore.mint.**floor**(*input*)

Rounds a tensor down to the closest integer element-wise.

$$out_i = \lfloor input_i \rfloor$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.1, 2.5, -1.5])
>>> output = mindspore.mint.floor(input)
>>> print(output)
[ 1.  2. -2.]
```

mindspore.mint.fmod

mindspore.mint.**fmod**(*input*, *other*) → *Tensor*

Computes the floating-point remainder of the division operation input/other.

$$out = input - n * other$$

Where n is $input/other$ with its fractional part truncated. The returned value has the same sign as *input* and is less than *other* in magnitude.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – the dividend.
- **other** (*Union[Tensor, Number]*) – the divisor.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-4., -3.5, 0, 3.5, 4]), mindspore.float32)
>>> output = mint.fmod(input, 2.5)
>>> print(output)
[-1.5 -1.   0.   1.   1.5]
```

mindspore.mint.frac

mindspore.mint.**frac**(*input*)

Calculates the fractional part of each element in the input.

$$out_i = input_i - \lfloor |input_i| \rfloor * sgn(input_i)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) – The input Tensor.

Returns

Tensor, has the same shape and type as input.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor([2, 4.2, -2.5], mindspore.float16)
>>> output = mint.frac(x)
>>> print(output)
  [ 0.      0.1992 -0.5   ]
```

mindspore.mint.lerp

mindspore.mint.**lerp**(*input*, *end*, *weight*) → *Tensor*

Perform a linear interpolation of two tensors input and end based on a float or tensor weight.

If *weight* is a tensor, the shapes of three inputs need to be broadcast; If *weight* is a float, the shapes of *input* and *end* need to be broadcast. If *weight* is a float and platform is Ascend, the types of *input* and *end* need to be float32.

Warning: This is an experimental API that is subject to change or deletion.

$$output_i = input_i + weight_i * (end_i - input_i)$$

Parameters

- **input** (Tensor) –The tensor with the starting points. Data type must be float16 or float32.
- **end** (Tensor) –The tensor with the ending points. Data type must be the same as *input*.
- **weight** (*Union[float, Tensor]*) –The weight for the interpolation formula. Must be a float scalar or a tensor with float16 or float32 data type.

Returns

Tensor, has the same type and shape as input *input*.

Raises

- **TypeError** –If *input* or *end* is not a tensor.
- **TypeError** –If *weight* is neither scalar(float) nor tensor.
- **TypeError** –If dtype of *input* or *end* is neither float16 nor float32.
- **TypeError** –If dtype of *weight* is neither float16 nor float32 when it is a tensor.
- **TypeError** –If *input* and *end* have different data types.
- **TypeError** –If *input*, *end* and *weight* have different data types when *weight* is a tensor.
- **ValueError** –If *end* could not be broadcast to a tensor with shape of *input*.
- **ValueError** –If *weight* could not be broadcast to tensors with shapes of *input* and *end* when it is a tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> start = Tensor(np.array([1., 2., 3., 4.]), mindspore.float32)
>>> end = Tensor(np.array([10., 10., 10., 10.]), mindspore.float32)
>>> output = mint.lerp(start, end, 0.5)
>>> print(output)
[5.5 6. 6.5 7. ]
```

mindspore.mint.log

mindspore.mint.**log**(*input*)

Compute the natural logarithm of the input tensor element-wise.

$$y_i = \log_e(x_i)$$

Warning: If the input value of operator Log is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.mint.log(x)
>>> print(output)
[0.        0.6931472 1.3862944]
```

mindspore.mint.log1p

mindspore.mint.**log1p**(*input*)

Compute the natural logarithm of (tensor + 1) element-wise.

$$out_i = \log_e(input_i + 1)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.mint.log1p(x)
>>> print(output)
[0.6931472 1.0986123 1.609438 ]
```

mindspore.mint.log2

mindspore.mint.**log2**(*input*)

Returns the logarithm to the base 2 of a tensor element-wise.

$$y_i = \log_2(x_i)$$

Warning:

- If the input value of operator Log2 is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

input (Tensor) – Input Tensor of any dimension. The value must be greater than 0.

Returns

Tensor, has the same shape as the *input*. If *input.dtype* is of integer or boolean type, the output dtype will be float32. Otherwise, the output dtype will be the same as *input.dtype*.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([3.0, 5.0, 7.0]), mindspore.float32)
>>> output = mint.log2(x)
>>> print(output)
[1.5849625 2.321928  2.807355 ]
```

mindspore.mint.log10

mindspore.mint.**log10**(*input*)

Returns the logarithm to the base 10 of a tensor element-wise.

$$y_i = \log_{10}(x_i)$$

Warning:

- This is an experimental API that is subject to change or deletion.
- If the input value of operator Log10 is within the range (0, 0.01] or [0.95, 1.05], the output accuracy may be affacted.

Parameters

input (Tensor) – Input Tensor of any dimension. The value must be greater than 0.

Returns

Tensor, has the same shape as the *input*, and the dtype changes according to the *input.dtype*.

- if *input.dtype* is in [float16, float32, float64, bfloat16], the output dtype is the same as the *input.dtype*.
- if *input.dtype* is integer or boolean type, the output dtype is float32.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([3.0, 5.0, 7.0]), mindspore.float32)
>>> output = mint.log10(x)
>>> print(output)
[0.47712136 0.69897    0.845098  ]
```

mindspore.mint.logaddexp

mindspore.mint.**logaddexp**(*input*, *other*)

Computes the logarithm of the sum of exponentiations of the inputs. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers.

$$out_i = \log(exp(input_i) + \exp(other_i))$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Input Tensor. The dtype of *input* must be float.
- **other** (Tensor) –Input Tensor. The dtype of *other* must be float. If the shape of *input* is not equal to the shape of *other*, they must be broadcastable to a common shape.

Returns

Tensor, with the same dtype as *input* and *other*.

Raises

- **TypeError** –If *input* or *other* is not a Tensor.
- **TypeError** –The dtype of *input* or *other* is not float.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x1 = Tensor(np.array([1, 2, 3]).astype(np.float16))
>>> x2 = Tensor(np.array(2).astype(np.float16))
>>> output = mint.logaddexp(x1, x2)
>>> print(output)
[2.312 2.693 3.312]
```

mindspore.mint.logaddexp2

mindspore.mint.**logaddexp2**(*input*, *other*)

Logarithm of the sum of exponentiations of the inputs in base of 2.

$$out_i = \log_2(2^{input_i} + 2^{other_i})$$

Parameters

- **input** (Tensor) –Input Tensor. The dtype of *input* must be float.
- **other** (Tensor) –Input Tensor. The dtype of *other* must be float. If the shape of *input* is not equal to the shape of *other*, they must be broadcastable to a common shape (which becomes the shape of the output).

Returns

Tensor, with the same dtype as *input* and *other*.

Raises

- **TypeError** –If *input* or *other* is not a Tensor.
- **TypeError** –The dtype of *input* or *other* is not float.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x1 = Tensor(np.array([1, 2, 3]).astype(np.float16))
>>> x2 = Tensor(np.array(2).astype(np.float16))
>>> output = mint.logaddexp2(x1, x2)
>>> print(output)
[2.586 3. 3.586]
```

mindspore.mint.logical_and

mindspore.mint.**logical_and**(*input*, *other*)

Compute the "logical AND" of two tensors element-wise.

$$out_i = input_i \wedge other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, bool]*) – The first input.
- **other** (*Union[Tensor, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_and(x, y)
>>> print(output)
[ True False False]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_and(x, y)
>>> print(output)
False
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_and(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
False
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_and(x, y)
>>> print(output)
[True False]
```

mindspore.mint.logical_not

mindspore.mint.**logical_not**(*input*)

Compute the "logical NOT" of the input tensor element-wise.

$$out_i = \neg input_i$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> output = mindspore.mint.logical_not(x)
>>> print(output)
[False  True False]
```

mindspore.mint.logical_or

mindspore.mint.**logical_or**(*input*, *other*)

Compute the "logical OR" of two tensors element-wise.

$$out_i = input_i \vee other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, bool]*) – The first input.

- **other** (*Union[Tensor, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_or(x, y)
>>> print(output)
[ True  True  True]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_or(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_or(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_or(x, y)
>>> print(output)
[True True]
```

mindspore.mint.logical_xor

mindspore.mint.**logical_xor**(*input*, *other*)

Compute the "logical XOR" of two tensors element-wise.

$$out_i = input_i \oplus other_i$$

Note:

- Broadcasting is supported.
- Support implicit type conversion.

Parameters

- **input** (Tensor) –The first input tensor.
- **other** (Tensor) –The second input tensor.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([True, False, True], mindspore.bool_)
>>> y = mindspore.tensor([True, True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_xor(x, y)
>>> print(output)
[False  True  True]
>>> x = mindspore.tensor(1, mindspore.bool_)
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_xor(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor(0, mindspore.bool_)
>>> output = mindspore.mint.logical_xor(x, y)
>>> print(output)
True
>>> x = True
>>> y = mindspore.tensor([True, False], mindspore.bool_)
>>> output = mindspore.mint.logical_xor(x, y)
>>> print(output)
[False  True]
```

mindspore.mint.mul

`mindspore.mint.`**`mul`**(*input*, *other*)

Multiply other value by input Tensor.

$$out_i = input_i \times other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs comply with the implicit type conversion rules to make the data types consistent.

Parameters

- **`input`** (*Union[Tensor, number.Number, bool]*) – The first input is a number.Number or a bool or a tensor whose data type is number or bool_.
- **`other`** (*Union[Tensor, number.Number, bool]*) – The second input, is a number.Number or a bool or a tensor whose data type is number or bool_.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

TypeError –If the type of *input*, *other* is not one of the following: Tensor, number.Number, bool.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([2, 6, 9]).astype(np.int32))
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> output = mint.mul(x, y)
>>> print(output)
[8. 30. 54.]
>>> # the data type of x is int32, the data type of y is float32,
>>> # and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

mindspore.mint.mv

mindspore.mint.**mv**(*input*, *vec*)

Multiply matrix *input* and vector *vec*. If *input* is a tensor with shape (N, M) and *vec* is a tensor with shape $(M,)$, The output is a 1-D tensor which shape is $(N,)$.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input matrix which shape is (N, M) and the rank must be 2-D.
- **vec** (Tensor) –The input vector which shape is $(M,)$ and the rank is 1-D.

Returns

Tensor, the shape is $(N,)$.

Raises

- **TypeError** –If *input* or *vec* is not a tensor.
- **TypeError** –If the dtype of *input* or *vec* is not float16 or float32.
- **TypeError** –If the dtypes of *input* and *vec* are different.
- **ValueError** –If the *input* is not a 2-D tensor or the *vec* is not a 1-D tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[3., 4.], [1., 6.], [1., 3.]]).astype(np.float32))
>>> vec = Tensor(np.array([1., 2.]).astype(np.float32))
>>> output = mint.mv(input, vec)
>>> print(output)
[11. 13. 7.]
```

mindspore.mint.nansum

mindspore.mint.**nansum**(*input*, *dim=None*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Computes sum of *input* over a given dimension, treating NaNs as zero.

> **Warning:** It is only supported on Atlas A2 Training Series Products. This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input Tensor.
- **dim** (*Union[int, tuple(int)], optional*) –The dimensions to sum. Dim must be in the range [-rank(input), rank(input)). Default: None, which indicates the sum of all elements in a tensor.
- **keepdim** (*bool, optional*) –Whether the output Tensor keeps dimensions or not. Default: False, indicating that no dimension is kept.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The dtype of output Tensor. Default: None.

Returns

Tensor, the sum of input *input* in the given dimension dim, treating NaNs as zero.

- If dim is None, keepdim is False, the output is a 0-D Tensor representing the sum of all elements in the input Tensor.
- If dim is int, set as 2, and keepdim is False, the shape of output is $(input_1, input_3, ..., input_R)$.
- If dim is tuple(int) or list(int), set as (2, 3), and keepdim is False, the shape of output is $(input_1, input_4, ..., input_R)$.

Raises

- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **TypeError** –If the dtype of *input* or *dtype* is complex type.
- **ValueError** –If *dim* is not in [-rank(input), rank(input)).

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[float("nan"), 2, 3], [1, 2, float("nan")]]), mindspore.float32)
>>> output1 = mint.nansum(x, dim=0, keepdim=False, dtype=mindspore.float32)
>>> output2 = mint.nansum(x, dim=0, keepdim=True, dtype=mindspore.float32)
>>> print(output1)
[1. 4. 3.]
>>> print(output2)
[[1. 4. 3.]]
```

mindspore.mint.nan_to_num

mindspore.mint.**nan_to_num**(*input*, *nan=None*, *posinf=None*, *neginf=None*)

Replace the *NaN*, positive infinity and negative infinity values in *input* with the specified values in *nan*, *posinf* and *neginf* respectively.

Warning: For Ascend, it is only supported on Atlas A2 Training Series Products. This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **nan** (*number, optional*) –The replace value of *NaN*. Default None.
- **posinf** (*number, optional*) –the value to replace *posinf* values with. Default None, replacing *posinf* with the maximum value supported by the data type of *input*.
- **neginf** (*number, optional*) –the value to replace *neginf* values with. Default None, replacing *neginf* with the minimum value supported by the data type of *input*.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([float('nan'), float('inf'), -float('inf'), 5.0], mindspore.
↪float32)
>>> output = mindspore.mint.nan_to_num(input)
>>> print(output)
[ 0.0000000e+00  3.4028235e+38 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.mint.nan_to_num(input, 1.0)
>>> print(output)
[ 1.0000000e+00  3.4028235e+38 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.mint.nan_to_num(input, 1.0, 2.0)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[ 1.0000000e+00  2.0000000e+00 -3.4028235e+38  5.0000000e+00]
>>> output = mindspore.mint.nan_to_num(input, 1.0, 2.0, 3.0)
>>> print(output)
[1.  2.  3.  5.0]
```

mindspore.mint.neg

mindspore.mint.**neg**(*input*)

Returns a tensor with negative values of the input tensor element-wise.

$$out_i = -input_i$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 2, -1, 2, 0, -3.5], mindspore.float32)
>>> output = mindspore.mint.neg(input)
>>> print(output)
[-1.  -2.   1.  -2.   0.   3.5]
```

mindspore.mint.negative

mindspore.mint.**negative**(*input*)

Alias for *mindspore.mint.neg()*.

Supported Platforms:

Ascend

mindspore.mint.pow

mindspore.mint.**pow**(*input*, *exponent*)

Calculates the *exponent* power of each element in *input*.

When *exponent* is a Tensor, the shapes of *input* and *exponent* must be broadcastable.

$$out_i = input_i^{exponent_i}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (*Union[Tensor, Number]*) –The first input is a Number or a tensor whose data type is number or bool_.
- **exponent** (*Union[Tensor, Number]*) –The second input is a Number or a tensor whose data type is number or bool_.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** –If types of *input* and *exponent* are bool.
- **TypeError** –The *input* is tensor and of type int or bool, while the *exponent* is negative int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> exponent = 3.0
>>> output = mint.pow(input, exponent)
>>> print(output)
[ 1.  8. 64.]
>>>
>>> input = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> exponent = Tensor(np.array([2.0, 4.0, 3.0]), mindspore.float32)
>>> output = mint.pow(input, exponent)
>>> print(output)
[ 1. 16. 64.]
```

mindspore.mint.polar

mindspore.mint.**polar**(*abs*, *angle*)

Converts polar coordinates to Cartesian coordinates.

Returns a complex tensor, its elements are Cartesian coordinates constructed with the polar coordinates which is specified by radial distance *abs* and polar angle *angle*.

$$y_i = abs_i * \cos(angle_i) + abs_i * \sin(angle_i) * j$$

Parameters

- **abs** (*Tensor, float*) –Radial distance.
- **angle** (*Tensor, float*) –Polar angle.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> abs = mindspore.tensor([1, 2], mindspore.float32)
>>> angle = mindspore.tensor([np.pi / 2, 5 * np.pi / 4], mindspore.float32)
>>> output = mindspore.mint.polar(abs, angle)
>>> print(output)
[ -4.3711388e-08+1.j         -1.4142137e+00-1.4142134j]
```

mindspore.mint.ravel

mindspore.mint.**ravel**(*input*)

Expand the multidimensional Tensor into 1D along the 0 axis direction.

Parameters

input (Tensor) – A tensor to be flattened.

Returns

Tensor, a 1-D tensor, containing the same elements of the input.

Raises

TypeError – If argument *input* is not Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.float32))
>>> output = mint.ravel(x)
>>> print(output)
[0. 1. 2. 1.]
>>> print(output.shape)
(4,)
```

mindspore.mint.reciprocal

mindspore.mint.**reciprocal**(*input*)

Returns reciprocal of a tensor element-wise.

$$out_i = \frac{1}{x_i}$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.mint.reciprocal(input)
>>> print(output)
[1.   0.5  0.25]
```

mindspore.mint.remainder

mindspore.mint.**remainder**(*input*, *other*) → *Tensor*

Computes the remainder of *input* divided by *other* element-wise. The result has the same sign as the divisor and its absolute value is less than that of *other*.

Supports broadcasting to a common shape and implicit type promotion.

```
remainder(input, other) == input - input.div(other, rounding_mode="floor") * other
```

Note: Complex inputs are not supported. At least one input need to be tensor, but not both are bool tensors.

Parameters

- **input** (*Union[Tensor, numbers.Number, bool]*) –The dividend is a numbers.Number or a bool or a tensor whose data type is number or bool_.
- **other** (*Union[Tensor, numbers.Number, bool]*) –The divisor is a numbers.Number or a bool or a tensor whose data type is number or bool_ when the dividend is a tensor. When the dividend is Scalar, the divisor must be a Tensor whose data type is number or bool_.

Returns

Tensor, with dtype promoted and shape broadcasted.

Raises

- **TypeError** –If *input* and *other* are not of types: (tensor, tensor), (tensor, number), (tensor, bool), (number, tensor) or (bool, tensor).

- **ValueError** –If *input* and *other* are not broadcastable.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]).astype(np.float32))
>>> y = Tensor(np.array([3.0, 2.0, 3.0]).astype(np.float64))
>>> output = mint.remainder(x, y)
>>> print(output)
[2.  1.  0.]
```

mindspore.mint.roll

mindspore.mint.**roll**(*input*, *shifts*, *dims=None*)

Roll the elements of a tensor along a dimension.

Parameters

- **input** (Tensor) –The input tensor.
- **shifts** (*Union[list(int), tuple(int), int]*) –The amount of element shifting.
- **dims** (*Union[list(int), tuple(int), int], optional*) –Specify the dimension to move. Default None , which means the input tensor will be flattened before computation, and the result will be reshaped back to the original input shape.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0, 1, 2, 3, 4], mindspore.float32)
>>> # case1: Parameter `shifts` is positive
>>> output = mindspore.mint.roll(input, shifts=2, dims=0)
>>> print(output)
[3. 4. 0. 1. 2.]
>>> # case2: Parameter `shifts` is negative
>>> output = mindspore.mint.roll(input, shifts=-2, dims=0)
>>> print(output)
[2. 3. 4. 0. 1.]
```

mindspore.mint.round

mindspore.mint.**round**(*input*, *, *decimals=0*)

Round elements of input to the nearest integer.

$$out_i \approx input_i$$

Note: The input data types supported by the Ascend platform include bfloat16 (Atlas training series products are not supported), float16, float32, float64, int32, and int64.

Parameters

input (Tensor) −The input tensor.

Keyword Arguments

decimals (int, *optional*) −Number of decimal places to round. If decimals is negative, it specifies the number of positions to the left of the decimal point. Default 0 .

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.round(mindspore.tensor([4.7, -2.3, 9.1, -7.7]))
>>> print(output)
[ 5. -2.  9. -8.]
>>> # Values equidistant from two integers are rounded towards the
>>> # the nearest even value (zero is treated as even)
>>> output = mindspore.mint.round(mindspore.tensor([-0.5, 0.5, 1.5, 2.5]))
>>> print(output)
[0. 0.  2.  2.]
>>> # A positive decimals argument rounds to the to that decimal place
>>> output = mindspore.mint.round(mindspore.tensor([0.1234567]), decimals=3)
>>> print(output)
[0.123]
>>> # A negative decimals argument rounds to the left of the decimal
>>> output = mindspore.mint.round(mindspore.tensor([1200.1234567]), decimals=-3)
>>> print(output)
[1000.]
```

mindspore.mint.rsqrt

mindspore.mint.**rsqrt**(*input*)

Compute reciprocal of square root of input tensor element-wise.

$$out_i = \frac{1}{\sqrt{input_i}}$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-0.0370,  0.2970,  1.5420, -0.9105])
>>> output = mindspore.mint.rsqrt(input)
>>> print(output)
[       nan 1.8349396  0.8053002        nan]
```

mindspore.mint.sigmoid

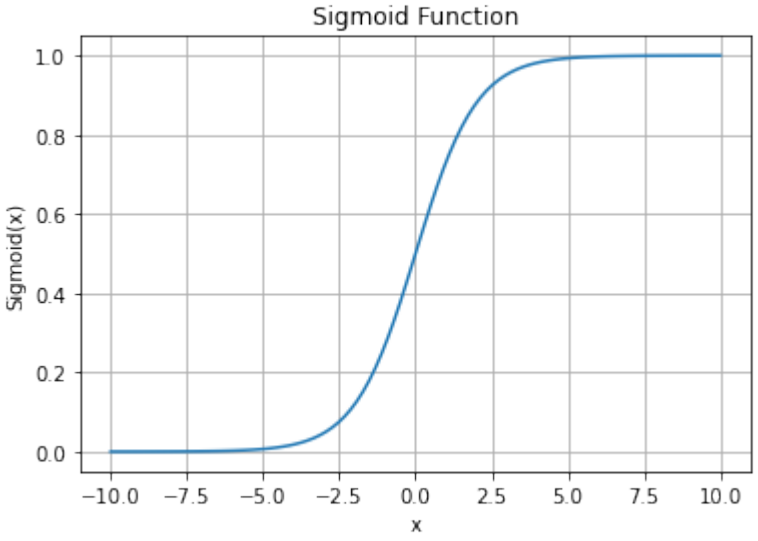
mindspore.mint.**sigmoid**(*input*)

Computes Sigmoid of input element-wise. The Sigmoid function is defined as:

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)}$$

where x_i is an element of x.

Sigmoid Function Graph:



Parameters

input (Tensor) – *input* is x in the preceding formula. Tensor of any dimension, the data type is float16, float32, float64, complex64 or complex128.

Returns

Tensor, with the same type and shape as the input.

Raises

- **TypeError** – If dtype of *input* is not float16, float32, float64, complex64 or complex128.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.sigmoid(input)
>>> print(output)
[0.7310586  0.880797   0.95257413 0.98201376 0.9933072 ]
```

mindspore.mint.sign

mindspore.mint.**sign**(*input*)

Return an element-wise indication of the sign of a number.

$$\text{out}_i = \begin{cases} -1 & \text{input}_i < 0 \\ 0 & \text{input}_i = 0 \\ 1 & \text{input}_i > 0 \end{cases}$$

Note: When the input is NaN and dtype is float64, the output of this operator is NaN.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[-1, 0, 2, 4, 6], [2, 3, 5, -6, 0]])
>>> output = mindspore.mint.sign(input)
>>> print(output)
[[-1  0  1  1  1]
 [ 1  1  1 -1  0]]
>>> mindspore.set_device(device_target="CPU")
>>> x = mindspore.tensor([[-1, 0, float('inf'), 4, float('nan')], [2, 3, float('-inf'), -6,␣
↪0]])
>>> output = mindspore.mint.sign(x)
>>> print(output)
[[-1.  0.  1.  1.  0.]
 [ 1.  1. -1. -1.  0.]]
```

mindspore.mint.sin

mindspore.mint.**sin**(*input*)

Compute sine of the input tensor element-wise.

$$output_i = \sin(input_i)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.mint.sin(input)
>>> print(output)
[0.58103514 0.27635565 0.4168708 0.58103514]
```

mindspore.mint.sinc

mindspore.mint.**sinc**(*input*)

Compute the normalized sinc of input.

$$out_i = \begin{cases} \frac{sin(\pi input_i)}{\pi input_i} & input_i \neq 0 \\ 1 & input_i = 0 \end{cases}$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.mint.sinc(input)
>>> print(output)
[0.47735003 0.8759357  0.7224278  0.47735003]
```

mindspore.mint.sinh

`mindspore.mint.`**`sinh`**(*input*)

Compute hyperbolic sine of the input element-wise.

$$output_i = \sinh(input_i)$$

Parameters

`input` (`Tensor`) −The input tensor.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.mint.sinh(input)
>>> print(output)
[0.6604918  0.28367308 0.44337422 0.6604918 ]
```

mindspore.mint.softmax

`mindspore.mint.`**`softmax`**(*input*, *dim*, *, *dtype=None*)

Alias for *`mindspore.mint.nn.functional.softmax()`*.

Supported Platforms:

`Ascend`

mindspore.mint.sqrt

mindspore.mint.**sqrt**(*input*)

Returns sqrt of a tensor element-wise.

$$out_i = \sqrt{input_i}$$

Parameters

input (Tensor) −The input tensor with a dtype of number.Number.

Returns

Tensor, has the same shape as the *input*.

Raises

TypeError −If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1.0, 4.0, 9.0]), mindspore.float32)
>>> output = mint.sqrt(input)
>>> print(output)
[1. 2. 3.]
```

mindspore.mint.square

mindspore.mint.**square**(*input*)

Return square of a tensor element-wise.

$$y_i = input_i^2$$

Parameters

input (Tensor) −The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> output = mindspore.mint.square(input)
>>> print(output)
[1. 4. 9.]
```

mindspore.mint.sub

mindspore.mint.**sub**(*input*, *other*, *, *alpha=1*) → *Tensor*

Subtracts scaled other value from self Tensor.

$$out_i = self_i - alpha \times other_i$$

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs and alpha comply with the implicit type conversion rules to make the data types consistent.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) – *input* is a number.Number or a bool or a tensor whose data type is number or bool_.
- **other** (*Union[Tensor, number.Number, bool]*) – *other* is a number.Number or a bool or a tensor whose data type is number or bool_.

Keyword Arguments

alpha (*number.Number, optional*) – A scaling factor applied to *other*, default 1.

Returns

Tensor with a shape that is the same as the broadcasted shape of the self *self* and *other*, and the data type is the one with higher precision or higher digits among the two inputs and alpha.

Raises

- **TypeError** – If the type of *other* or *alpha* is not one of the following: Tensor, number.Number, bool.
- **TypeError** – If *alpha* is of type float but *input* and *other* are not of type float.
- **TypeError** – If *alpha* is of type bool but *input* and *other* are not of type bool.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> y = Tensor(1, mindspore.int32)
>>> alpha = 0.5
>>> output = mint.sub(x, y, alpha=alpha)
>>> print(output)
[3.5 4.5 5.5]
>>> # the data type of x is float32, the data type of y is int32,
>>> # alpha is a float, and the output is the data format of higher precision float32.
>>> print(output.dtype)
Float32
```

mindspore.mint.t

mindspore.mint.**t**(*input*)

Transpose the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor, transpose 2D tensor, return 1D tensor as it is.

Raises

- **ValueError** – If the dimension of *input* is greater than 2.
- **ValueError** – If *input* is empty.
- **TypeError** – If *input* is not a tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[1, 2, 3], [4, 5, 6]]), mindspore.float32)
>>> output = mint.t(input)
>>> print(output)
[[ 1. 4.]
 [ 2. 5.]
 [ 3. 6.]]
```

mindspore.mint.tan

mindspore.mint.**tan**(*input*)

Computes tangent of *input* element-wise.

$$out_i = \tan(input_i)$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.tan(mindspore.tensor([-1.0, 0.0, 1.0]))
>>> print(output)
[-1.5574077 0. 1.5574077]
```

mindspore.mint.tanh

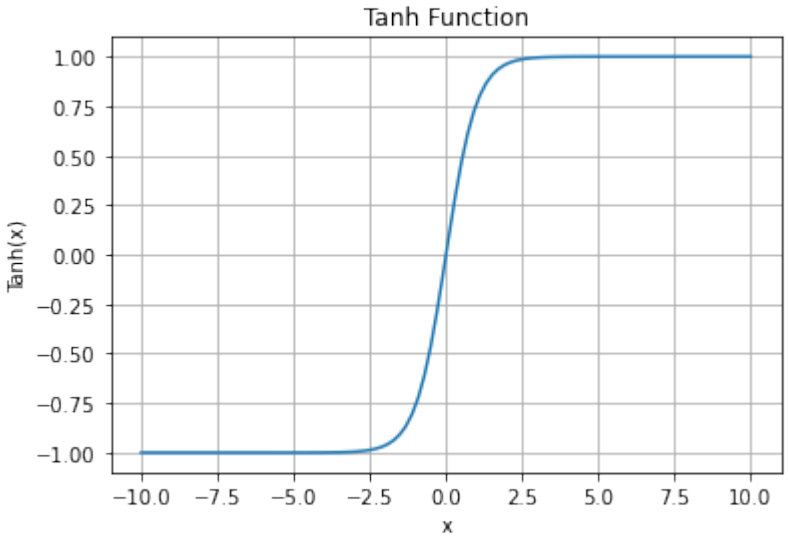
mindspore.mint.**tanh**(*input*)

Computes hyperbolic tangent of input element-wise. The Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Tanh Activation Function Graph:



Parameters

input (Tensor) –Input of Tanh.

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.tanh(input)
>>> print(output)
[0.7615941 0.9640276 0.9950547 0.9993293 0.9999092]
```

mindspore.mint.trunc

mindspore.mint.**trunc**(*input*)

Returns a tensor with the truncated integer values of the elements of the input tensor.

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> output = mindspore.mint.trunc(mindspore.tensor([3.4742, 0.5466, -0.8008, -3.9079]))
>>> print(output)
[3. 0. 0. -3.]
```

mindspore.mint.xlogy

mindspore.mint.**xlogy**(*input*, *other*) → *Tensor*

Computes the first input multiplied by the logarithm of second input element-wise. Returns zero when *input* is zero.

$$out_i = input_i \log other_i$$

Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, the shapes of them could be broadcast.

Parameters

- **input** (*Union[Tensor, numbers.Number, bool]*) – The first input is a numbers.Number or a bool or a tensor whose data type is number or bool_.
- **other** (*Union[Tensor, numbers.Number, bool]*) – The second input is a numbers.Number or a bool or a tensor whose data type is number or bool when the first input is a tensor. When the first input is Scalar, the second input must be a Tensor whose data type is number or bool.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** – If *input* and *other* is not a numbers.Number or a bool or a Tensor.
- **ValueError** – If *input* could not be broadcast to a tensor with shape of *other*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-5, 0, 4]), mindspore.float32)
>>> other = Tensor(np.array([2, 2, 2]), mindspore.float32)
>>> output = mint.xlogy(input, other)
>>> print(output)
[-3.465736   0.        2.7725887]
```

4.3.2 Reduction Operations

API Name	Description	Supported Platforms
mindspore.mint.amax	Computes the maximum value of of all elements along the specified *dim* dimension of the *input*, and retains the dimension based on the *keepdim* parameter.	Ascend
mindspore.mint.amin	Computes the minimum value of of all elements along the specified *dim* dimension of the *input*, and retains the dimension based on the *keepdim* parameter.	Ascend

continues on next page

Table 5 – continued from previous page

mindspore.mint.argmax	Return the indices of the maximum values of a tensor.	Ascend
mindspore.mint.argmin	Return the indices of the minimum values of a tensor across a dimension.	Ascend
mindspore.mint.argsort	Sorts the input tensor along the given dimension in specified order and return the sorted indices.	Ascend
mindspore.mint.all	Tests if all element in *input* evaluates to *True*.	Ascend
mindspore.mint.any	Tests if any element in *input* evaluates to *True* along the given axes.	Ascend
mindspore.mint.cumprod	Return the cumulative product along the given dimension of the tensor.	Ascend
mindspore.mint.histc	Computes the histogram of a tensor.	Ascend
mindspore.mint.logsumexp	Computes the logarithm of the sum of exponentiations of all elements along the specified *dim* dimension of the *input* (with numerical stabilization), and retains the dimension based on the *keepdim* parameter.	Ascend
mindspore.mint.max	Returns the maximum value of the input tensor.	Ascend
mindspore.mint.mean	Reduces all dimension of a tensor by averaging all elements.	Ascend
mindspore.mint.median	Output the median on the specified dimension `dim` and its corresponding index.	Ascend
mindspore.mint.min	Returns the minimum value of the input tensor.	Ascend
mindspore.mint.norm	Returns the matrix norm or vector norm of a given tensor.	Ascend
mindspore.mint.prod	Multiplying all elements of input.	Ascend
mindspore.mint.sum	Calculate sum of all elements in Tensor.	Ascend
mindspore.mint.std	Calculates the standard deviation over the dimensions specified by *dim*.	Ascend
mindspore.mint.std_mean	By default, return the standard deviation and mean of each dimension in Tensor.	Ascend
mindspore.mint.unique	Returns the unique elements of input tensor.	Ascend
mindspore.mint.var	Calculates the variance over the dimensions specified by *dim*.	Ascend
mindspore.mint.var_mean	By default, return the variance and mean of each dimension in Tensor.	Ascend

mindspore.mint.amax

mindspore.mint.**amax**(*input*, *dim=()*, *keepdim=False*)

Computes the maximum value of of all elements along the specified *dim* dimension of the *input*, and retains the dimension based on the *keepdim* parameter.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – Input tensor.
- **dim** (*Union[int, tuple(int), list(int)], optional*) – The dimension to be reduced, the value should be within *[-len(input.shape), len(input.shape) - 1]*, when the *dim* is *()*, all dimensions are reduced, default: `()`.

- **keepdim** (*bool, optional*) –Whether the output tensor retains the dimension *dim*, default: `False`.

Returns

Tensor, has same type as *input*, and the shape changes according to the values of *dim* and *keepdim*.

- If *dim* is *()*, and *keepdim* is False, the output is a 0-D tensor representing the maximum value of all elements in the *input* tensor.
- If *dim* is *1*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[2], ..., input.shape[n])$.
- If *dim* is *(1, 2)*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[3], ..., input.shape[n])$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not an int or tuple(int) or list(int).
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If the value of any elements of *dim* is not in the range *[-len(input.shape), len(input.shape) - 1]*.
- **RuntimeError** –If any element of *dim* is repeated.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = mint.amax(x, 1, keepdim=True)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by the maximum value of all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),
... mindspore.float32)
>>> output = mint.amax(x)
>>> print(output)
9.0
>>> print(output.shape)
()
>>> # case 2: Reduces a dimension along axis 0.
>>> output = mint.amax(x, 0, True)
>>> print(output)
[[[7. 7. 7. 7. 7. 7.]
  [8. 8. 8. 8. 8. 8.]
  [9. 9. 9. 9. 9. 9.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = mint.amax(x, 1, True)
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[[3. 3. 3. 3. 3. 3.]]
 [[6. 6. 6. 6. 6. 6.]]
 [[9. 9. 9. 9. 9. 9.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = mint.amax(x, 2, True)
>>> print(output)
[[[1.]
  [2.]
  [3.]]
 [[4.]
  [5.]
  [6.]]
 [[7.]
  [8.]
  [9.]]]
```

mindspore.mint.amin

mindspore.mint.**amin**(*input*, *dim=()*, *keepdim=False*)

Computes the minimum value of of all elements along the specified *dim* dimension of the *input*, and retains the dimension based on the *keepdim* parameter.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Input tensor.
- **dim** (*Union[int, tuple(int), list(int)], optional*) –The dimension to be reduced, the value should be within *[-len(input.shape), len(input.shape) - 1]*, when the *dim* is *()*, all dimensions are reduced, default: `()`.
- **keepdim** (*bool, optional*) –Whether the output tensor retains the dimension *dim*, default: `False`.

Returns

Tensor, has same type as *input*, and the shape changes according to the values of *dim* and *keepdim*.

- If *dim* is *()*, and *keepdim* is False, the output is a 0-D tensor representing the minimum value of all elements in the *input* tensor.
- If *dim* is *1*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[2], ..., input.shape[n])$.
- If *dim* is *(1, 2)*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[3], ..., input.shape[n])$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not an int or tuple(int) or list(int).
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If the value of any elements of *dim* is not in the range *[-len(input.shape), len(input.shape) - 1]*.
- **RuntimeError** –If any element of *dim* is repeated.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = mint.amin(x, 1, keepdim=True)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by the minimum value of all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),
... mindspore.float32)
>>> output = mint.amin(x)
>>> print(output)
1.0
>>> print(output.shape)
()
>>> # case 2: Reduces a dimension along axis 0.
>>> output = mint.amin(x, 0, True)
>>> print(output)
[[[1. 1. 1. 1. 1. 1.]
  [2. 2. 2. 2. 2. 2.]
  [3. 3. 3. 3. 3. 3.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = mint.amin(x, 1, True)
>>> print(output)
[[[1. 1. 1. 1. 1. 1.]]
 [[4. 4. 4. 4. 4. 4.]]
 [[7. 7. 7. 7. 7. 7.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = mint.amin(x, 2, True)
>>> print(output)
[[[1.]
  [2.]
  [3.]]
 [[4.]
  [5.]
  [6.]]
 [[7.]
  [8.]
  [9.]]]
```

mindspore.mint.argmax

mindspore.mint.**argmax**(*input*) → *Tensor*

Return the indices of the maximum values of a tensor.

Parameters

input (Tensor) – Input tensor.

Returns

Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([[1, 20, 5], [67, 8, 9], [130, 24, 15]]).astype(np.float32))
>>> output = mint.argmax(x)
>>> print(output)
6
```

mindspore.mint.**argmax**(*input*, *dim*, *keepdim=False*) → *Tensor*

Return the indices of the maximum values of a tensor across a dimension.

Parameters

- **input** (Tensor) – Input tensor.
- **dim** (*int*) – The dimension to reduce.
- **keepdim** (*bool, optional*) – Whether the output tensor retains the specified dimension. Default: False .

Returns

Tensor, indices of the maximum values across a dimension.

Raises

- **TypeError** – If *keepdim* is not bool.
- **ValueError** – If *dim* is out of range.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([[1, 20, 5], [67, 8, 9], [130, 24, 15]]).astype(np.float32))
>>> output = mint.argmax(x, dim=-1)
>>> print(output)
[1 0 0]
```

mindspore.mint.argmin

mindspore.mint.**argmin**(*input*, *dim=None*, *keepdim=False*)

Return the indices of the minimum values of a tensor across a dimension.

Parameters

- **input** (Tensor) –Input tensor.
- **dim** (*Union[int, None], optional*) –Specify the axis for calculation. If *dim* is None , the indices of the minimum value within the flattened input will be returned. Default: None .
- **keepdim** (*bool, optional*) –Whether the output tensor retains the specified dimension. Ignored if *dim* is None. Default: False .

Returns

Tensor, indices of the minimum values of the input tensor across a dimension.

Raises

- **TypeError** –If *keepdim* is not bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([[1, 20, 5], [67, 8, 9], [130, 24, 15]]).astype(np.float32))
>>> output = mint.argmin(x, dim=-1)
>>> print(output)
[0 1 2]
```

mindspore.mint.argsort

mindspore.mint.**argsort**(*input*, *dim=- 1*, *descending=False*, *stable=False*)

Sorts the input tensor along the given dimension in specified order and return the sorted indices.

> **Warning:** This is an experimental optimizer API that is subject to change.

Parameters

- **input** (Tensor) – The input tensor to sort.
- **dim** (*int, optional*) – The dim to sort along. Default: -1 , means the last dimension. The Ascend backend only supports sorting the last dimension.
- **descending** (*bool, optional*) – The sort order. If *descending* is True then the elements are sorted in descending order by value. Otherwise sort in ascending order. Default: False .
- **stable** (*bool, optional*) – Whether to use stable sorting algorithm. Default: False.

Returns

Tensor, the indices of sorted input tensor. Data type is int64.

Raises

- **ValueError** – If *dim* is out of range.
- **TypeError** – If dtype of *dim* is not int32.
- **TypeError** – If dtype of *descending* is not bool.
- **TypeError** – If dtype of *stable* is not bool.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> sort = mint.argsort(x)
>>> print(sort)
[[2 1 0]
 [2 0 1]
 [0 1 2]]
```

mindspore.mint.all

mindspore.mint.**all**(*input*) → *Tensor*

Tests if all element in *input* evaluates to *True*.

Parameters

input (Tensor) –The input Tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[True, False], [True, True]])
>>> output = mindspore.mint.all(input)
>>> print(output)
False
```

mindspore.mint.**all**(*input*, *dim*, *keepdim=False*) → *Tensor*

Tests if all element in *input* evaluates to *True* along the given axes.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*Union[int, tuple(int), list(int), Tensor]*) –The dimensions to reduce. If None , all dimensions are reduced. Default None .
- **keepdim** (*bool, optional*) –Whether the output tensor has dim retained or not. Default False .

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1: Reduces a dimension along dim 1, with keepdim False.
>>> mindspore.mint.all(input, dim=1)
Tensor(shape=[2], dtype=Bool, value= [False,  True])
>>>
>>> # case 2: Reduces a dimension along dim (0, 1), with keepdim False.
>>> mindspore.mint.all(input, dim=(0,1))
Tensor(shape=[], dtype=Bool, value= False)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> # case 3: Reduces a dimension along dim [0, 1], with keepdim True.
>>> mindspore.mint.all(input, dim=[0,1], keepdim=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[False]])
```

mindspore.mint.any

mindspore.mint.**any**(*input*, *dim=None*, *keepdim=False*)

Tests if any element in *input* evaluates to *True* along the given axes.

Parameters

- **input** (Tensor) – The input tensor.
- **dim** (*Union[int, tuple(int), list(int), Tensor], optional*) – The dimensions to reduce. If None , all dimensions are reduced. Default None .
- **keepdim** (*bool, optional*) – Whether the output tensor has dim retained or not. Default False .

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[True, False], [True, True]])
>>>
>>> # case 1:  By default, mindspore.mint.any tests along all the axes.
>>> mindspore.mint.any(input)
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 2: Reduces a dimension along dim 1, with keepdim False.
>>> mindspore.mint.any(input, dim=1)
Tensor(shape=[2], dtype=Bool, value= [ True,  True])
>>>
>>> # case 3: Reduces a dimension along dim (0, 1), with keepdim False.
>>> mindspore.mint.any(input, dim=(0,1))
Tensor(shape=[], dtype=Bool, value= True)
>>>
>>> # case 4: Reduces a dimension along dim [0, 1], with keepdim True.
>>> mindspore.mint.any(input, dim=[0,1], keepdim=True)
Tensor(shape=[1, 1], dtype=Bool, value=
[[ True]])
```

mindspore.mint.cumprod

mindspore.mint.**cumprod**(*input*, *dim*, *dtype=None*)

Return the cumulative product along the given dimension of the tensor.

$$y_i = x_1 * x_2 * x_3 * ... * x_i$$

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*int*) –Specify the dimension for computation.
- **dtype** (*mindspore.dtype*, optional) –The data type returned. Default None .

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([[1, 2, 3],
...                           [4, 5, 6]])
>>> mindspore.mint.cumprod(input, dim=0)
Tensor(shape=[2, 3], dtype=Int64, value=
[[ 1,  2,  3],
 [ 4, 10, 18]])
>>> mindspore.mint.cumprod(input, dim=1)
Tensor(shape=[2, 3], dtype=Int64, value=
[[  1,   2,   6],
 [  4,  20, 120]])
```

mindspore.mint.histc

mindspore.mint.**histc**(*input*, *bins=100*, *min=0*, *max=0*)

Computes the histogram of a tensor.

The elements are sorted into equal width bins between *min* and *max*. If *min* and *max* are both zero, the minimum and maximum values of the data are used.

Elements lower than min or higher than max are ignored.

> **Warning:** This is an experimental API that is subject to change or deletion. If input is int64, valid values fit within int32; exceeding this may cause precision errors.

Parameters

- **input** (Tensor) –the input tensor.
- **bins** (*int, optional*) –Number of histogram bins, optional. If specified, must be positive. Default: 100 .
- **min** (*int, float, optional*) –the lower end of the range (inclusive), optional. Default: 0 .

- **max** (*int, float, optional*) –the upper end of the range (inclusive), optional. Default: `0` .

Returns

A 1-D Tensor, has the same type as *input* with the shape $(bins,)$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *input* datatype is not in support list.
- **TypeError** –If attr *min* or *max* is not float or int.
- **TypeError** –If attr *bins* is not int.
- **ValueError** –If attr value *min* > *max*.
- **ValueError** –If attr *bins* <= 0.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> x = Tensor([1., 2, 1])
>>> y = mint.histc(x, bins=4, min=0, max=3)
>>> print(y)
[0 2 1 0]
```

mindspore.mint.logsumexp

`mindspore.mint.`**`logsumexp`**(*input*, *dim*, *keepdim=False*)

Computes the logarithm of the sum of exponentiations of all elements along the specified *dim* dimension of the *input* (with numerical stabilization), and retains the dimension based on the *keepdim* parameter.

$$logsumexp(input) = \log(\sum(e^{input - input_{max}})) + input_{max}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) –Input Tensor.
- **dim** (*Union[int, tuple(int), list(int)], optional*) –The dimension to be reduced (the value should be within *[0, len(input.shape) - 1]*), when the *dim* is *()*, all dimensions are reduced.
- **keepdim** (*bool, optional*) –Whether the output tensor retains the dimension *dim*, default: *False*.

Returns

Tensor, the dtype changes according to the *input.dtype*, and the shape changes according to the values of *dim* and *keepdim*.

- If *input.dtype* is in [float16, float32, bfloat16], the output dtype is the same as the *input.dtype*.

- If *input.dtype* is an integer or boolean type, the output dtype is float32.
- If *dim* is (), and *keepdim* is False, the output is a 0-D tensor representing the logarithm of the sum of exponentiations of all elements in the *input* tensor.
- If *dim* is *1*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[2], ..., input.shape[n])$.
- If *dim* is *(1, 2)*, and *keepdim* is False, the shape of output is $(input.shape[0], input.shape[3], ..., input.shape[n])$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not one of: bool, int8, int16, int32, int64, uint8, float16, float32, bfloat16.
- **TypeError** –If *dim* is not an int or tuple(int) or list(list).
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If the value of any elements of *dim* is not in the range *[0, len(input.shape) - 1]*.
- **RuntimeError** –If any element of *dim* is repeated.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = mint.logsumexp(x, 1, keepdim=True)
>>> print(output.shape)
(3, 1, 5, 6)
```

mindspore.mint.max

mindspore.mint.**max**(*input*) → *Tensor*

Returns the maximum value of the input tensor.

Parameters

input (Tensor) –The input tensor.

Returns

Scalar Tensor with the same dtype as *input*, the maximum value of the input.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output = mint.max(x)
>>> print(output)
0.7
```

mindspore.mint.**max**(*input*, *dim*, *keepdim=False*)

Calculates the maximum value along with the given dim for the input tensor, and returns the maximum values and indices.

Parameters

- **input** (Tensor) –The input tensor, can be any dimension. Set the shape of input tensor as $(input_1, input_2, ..., input_N)$, Complex tensor is not supported.
- **dim** (*int*) –The dimension to reduce.
- **keepdim** (*bool, optional*) –Whether to reduce dimension, if `True` the output will keep the same dimension as the *input* , the output will reduce dimension if `False`. Default: `False`.

Returns

tuple (Tensor), tuple of 2 tensors, containing the maximum value of the self tensor along the given dimension *dim* and the corresponding index.

- **values** (Tensor) - The maximum value of input tensor, with the same shape as *index*, and same dtype as *input*.
- **index** (Tensor) - The index for the maximum value of the input tensor, with dtype int64. If *keepdim* is `True` , the shape of output tensors is $(input_1, input_2, ..., input_{dim-1}, 1, input_{dim+1}, ..., input_N)$. Otherwise, the shape is $(input_1, input_2, ..., input_{dim-1}, input_{dim+1}, ..., input_N)$.

Raises

- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **TypeError** –If *dim* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output, index = mint.max(x, 0, keepdim=True)
>>> print(output, index)
[0.7] [3]
```

mindspore.mint.**max**(*input*, *other*) → *Tensor*

For details, please refer to *mindspore.mint.maximum()*.

mindspore.mint.mean

mindspore.mint.**mean**(*input*, *, *dtype=None*) → *Tensor*

Reduces all dimension of a tensor by averaging all elements.

Parameters

input (*Tensor[Number]*) –The input tensor. The dtype of the tensor to be reduced is number. $(N, *)$ where $*$ means, any number of additional dimensions.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: None .

Returns

Tensor.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[6, 6, 6, 6, 6, 6], [8, 8, 8, 8, 8, 8], [10, 10, 10, 10, 10, 10]]]),
... mindspore.float32)
>>> output = mint.mean(x)
>>> print(output)
5.0
>>> print(output.shape)
()
```

mindspore.mint.**mean**(*input*, *dim*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Reduces all dimension of a tensor by averaging all elements in the dimension, by default. And reduce a dimension of *input* along the specified *dim*. *keepdim* determines whether the dimensions of the output and input are the same.

Note: The *dim* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **input** (*Tensor[Number]*) –The input tensor. The dtype of the tensor to be reduced is number. $(N, *)$ where $*$ means, any number of additional dimensions.
- **dim** (*Union[int, tuple(int), list(int), Tensor]*) –The dimensions to reduce. Only constant value is allowed. Assume the rank of *input* is r, and the value range is [-r,r).
- **keepdim** (*bool*) –If True , keep these reduced dimensions and the length is 1. If False , don't keep these dimensions. Default: False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The desired data type of returned Tensor. Default: `None` .

Returns

Tensor.

- If *dim* is int, set as 1, and *keepdim* is `False` , the shape of output is $(input_0, input_2, ..., input_R)$.
- If *dim* is tuple(int) or list(int), set as (1, 2), and *keepdim* is `False` , the shape of output is $(input_0, input_3, ..., input_R)$.
- If *dim* is 1-D Tensor, set as [1, 2], and *keepdim* is `False` , the shape of output is $(input_0, input_3, ..., input_R)$.

Raises

- **TypeError** −If *input* is not a Tensor.
- **TypeError** −If *dim* is not one of the following: int, tuple, list or Tensor.
- **TypeError** −If *keepdim* is not a bool.
- **ValueError** −If *dim* is out of range.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[6, 6, 6, 6, 6, 6], [8, 8, 8, 8, 8, 8], [10, 10, 10, 10, 10, 10]]]),
... mindspore.float32)
>>> output = mint.mean(x, 0, True)
>>> print(output)
[[[4. 4. 4. 4. 4. 4.]
  [5. 5. 5. 5. 5. 5.]
  [6. 6. 6. 6. 6. 6.]]]
```

mindspore.mint.median

`mindspore.mint.`**median**(*input*, *dim=None*, *keepdim=False*)

Output the median on the specified dimension `dim` and its corresponding index. If `dim` is None, calculate the median of all elements in the Tensor.

Parameters

- **input** (`Tensor`) −A Tensor of any dimension whose data type is uint8, int16, int32, int64, float16 or float32.
- **dim** (*int, optional*) −Specify the axis for calculation. Default: `None` .
- **keepdim** (*bool, optional*) −Whether the output tensor need to retain `dim` dimension or not. Default: `False` .

Returns

- y (Tensor) - Output median, with the same data type as `input` .

- If `dim` is `None` , `y` only has one element.
- If `keepdim` is `True` , the `y` has the same shape as the `input` except the shape of `y` in dimension *dim* is size 1.
- Otherwise, the `y` lacks *dim* dimension than input.

- indices (Tensor) - The index of the median. Shape is consistent with `y` , with a data type of int64.

Raises

- **TypeError** –If dtype of `input` is not one of the following: uint8, int16, int32, int64, float16 or float32.
- **TypeError** –If input `input` is not a Tensor.
- **TypeError** –If `dim` is not an int.
- **TypeError** –If `keepdim` is not a bool.
- **ValueError** –If `dim` is not in range of [-x.dim, x.dim-1].

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[0.57, 0.11, 0.21],[0.38, 0.50, 0.57], [0.36, 0.16, 0.44]]).
↪astype(np.float32))
>>> y = mint.median(x, dim=0, keepdim=False)
>>> print(y)
(Tensor(shape=[3], dtype=Float32, value= [ 3.79999995e-01,  1.59999996e-01,  4.39999998e-
↪01]),
Tensor(shape=[3], dtype=Int64, value= [1, 2, 2]))
```

mindspore.mint.min

mindspore.mint.**min**(*input*) → *Tensor*

Returns the minimum value of the input tensor.

Parameters

input (Tensor) –The input tensor.

Returns

Scalar Tensor with the same dtype as *input*, the minimum value of the input.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output = mint.min(x)
>>> print(output)
0.0
```

mindspore.mint.**min**(*input*, *dim*, *keepdim=False*) → *Tensor*

Calculates the minimum value along with the given dim for the input tensor, and returns the minimum values and indices.

Parameters

- **input** (Tensor) –$(input_1, input_2, ..., input_N)$, Complex tensor is not supported.
- **dim** (*int*) –The dimension to reduce.
- **keepdim** (*bool, optional*) –Whether to reduce dimension, if `True` the output will keep the same dimension as the input, the output will reduce dimension if `False`. Default: `False`.

Returns

tuple (Tensor), tuple of 2 tensors, containing the minimum value of the self tensor along the given dimension *dim* and the corresponding index.

- **values** (Tensor) - The minimum value of input tensor, with the same shape as *index*, and same dtype as *input*.
- **index** (Tensor) - The index for the minimum value of the input tensor, with dtype int64. If *keepdim* is `True` , the shape of output tensors is $(input_1, input_2, ..., input_{dim-1}, 1, input_{dim+1}, ..., input_N)$. Otherwise, the shape is $(input_1, input_2, ..., input_{dim-1}, input_{dim+1}, ..., input_N)$.

Raises

- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **TypeError** –If *dim* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> output, index = mint.min(x, 0, keepdim=True)
>>> print(output, index)
[0.0] [0]
```

mindspore.mint.**min**(*input*, *other*) → *Tensor*

For details, please refer to *mindspore.mint.minimum()*.

mindspore.mint.norm

mindspore.mint.**norm**(*input*, *p='fro'*, *dim=None*, *keepdim=False*, *, *dtype=None*)

Returns the matrix norm or vector norm of a given tensor.

p is the calculation mode of norm. The following norm modes are supported.

p	norm for matrices	norm for vectors
'fro'	Frobenius norm	−not supported −
'nuc'	nuclear norm	−not supported −
other *int* or *float*	−not supported −	$sum(abs(x)^p)^{(1/p)}$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −The shape is $(*)$ or $(*, m, n)$ where $*$ means, any number of additional dimensions.
- **p** (*Union[bool, int, float, inf, -inf, 'fro', 'nuc'], optional*) −norm's mode. refer to the table above for behavior. Default: `fro` .
- **dim** (*Union[int, List(int), Tuple(int)], optional*) −calculate the dimension of vector norm or matrix norm. Default: `None` .
- **keepdim** (*bool, optional*) −whether the output Tensor retains the original dimension. Default: `False` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) −When set, *input* will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. Default: `None` .

Returns

Tensor, the result of norm calculation on the specified dimension, *dim*.

Raises

- **TypeError** −If *input* is not a Tensor.
- **ValueError** −If *dim* is out of range.
- **TypeError** −If *dim* is neither an int nor a tuple of int.
- **ValueError** −If two elements of *dim* is same after normalized.
- **ValueError** −If any elements of *dim* is out of range.

Note: Dynamic shape, Dynamic rank and mutable input is not supported in graph mode (mode=mindspore.GRAPH_MODE).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint, ops
>>> data_range = ops.arange(-13, 13, dtype=ms.float32)
>>> x = data_range[data_range != 0]
>>> y = x.reshape(5, 5)
>>> print(mint.norm(x, 2.0))
38.327538
```

mindspore.mint.prod

mindspore.mint.**prod**(*input*, *, *dtype=None*) → *Tensor*

Multiplying all elements of input.

Parameters

input (Tensor[*Number*]) – The input tensor. The dtype of the tensor to be reduced is number. $(N, *)$ where $*$ means, any number of additional dimensions.

Keyword Arguments

dtype (*mindspore.dtype*, optional) – The desired data type of returned Tensor. Default: None .

Returns

Tensor.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = mint.prod(x)
>>> print(output)
2.2833798e+33
>>> print(output.shape)
()
```

mindspore.mint.**prod**(*input*, *dim*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Reduces a dimension of a tensor by multiplying all elements in the dimension, by default. And also can reduce a dimension of *input* along the *dim*. Determine whether the dimensions of the output and input are the same by controlling *keepdim*.

Parameters

- **input** (*Tensor[Number]*) –The input tensor. The dtype of the tensor to be reduced is number. $(N, *)$ where $*$ means, any number of additional dimensions.
- **dim** (*int*) –The dimensions to reduce. Only constant value is allowed. Assume the rank of x is r, and the value range is [-r,r).
- **keepdim** (*bool*) –If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: `None` .

Returns

Tensor.

- If *dim* is int, set as 1, and *keepdim* is `False` , the shape of output is $(input_0, input_2, ..., input_R)$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not int.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = mint.prod(x, 0, True)
>>> print(output)
[[[ 28.  28.  28.  28.  28.  28.]
  [ 80.  80.  80.  80.  80.  80.]
  [162. 162. 162. 162. 162. 162.]]]
```

mindspore.mint.sum

mindspore.mint.**sum**(*input*, *, *dtype=None*) → *Tensor*

Calculate sum of all elements in Tensor.

Parameters

input (Tensor) –The input tensor.

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: None .

Returns

A Tensor, sum of all elements in *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mstype.float32)
>>> out = mint.sum(x)
>>> print(out)
270.0
```

mindspore.mint.**sum**(*input*, *dim*, *keepdim=False*, *, *dtype=None*) → *Tensor*

Calculate sum of Tensor elements over a given dim.

Note: The *dim* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **input** (Tensor) –The input tensor.
- **dim** (*Union[int, tuple(int), list(int), Tensor]*) –Dimensions along which a sum is performed. If the *dim* is a tuple or list of ints, a sum is performed on all the dimensions specified in the tuple. Must be in the range $[-input.ndim, input.ndim)$.
- **keepdim** (*bool*) –Whether the output tensor has *dim* retained or not. If True , keep these reduced dimensions and the length is 1. If False , don't keep these dimensions. Default: False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired data type of returned Tensor. Default: None .

Returns

A Tensor, sum of elements over a given *dim* in *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *dim* is not an int, tulpe(int), list(int) or Tensor.
- **ValueError** – If *dim* is not in the range $[-input.ndim, input.ndim)$.
- **TypeError** – If *keepdim* is not a bool.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mstype.float32)
>>> out = mint.sum(x)
>>> print(out)
270.0
>>> out = mint.sum(x, dim=2)
>>> print(out)
[[ 6. 12. 18.]
 [24. 30. 36.]
 [42. 48. 54.]]
>>> out = mint.sum(x, dim=2, keepdim=True)
>>> print(out)
[[[ 6.]
  [12.]
  [18.]]
 [[24.]
  [30.]
  [36.]]
 [[42.]
  [48.]
  [54.]]]
```

mindspore.mint.std

mindspore.mint.**std**(*input*, *dim=None*, *, *correction=1*, *keepdim=False*)

Calculates the standard deviation over the dimensions specified by *dim*. *dim* can be a single dimension, list of dimensions, or None to reduce over all dimensions.

The standard deviation (σ) is calculated as:

$$\sigma = \sqrt{\frac{1}{N - \delta N}\sum_{j-1}^{N-1}\left(self_{ij} - \overline{x_i}\right)^2}$$

where x is the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −The tensor used to calculate the standard deviation.
- **dim** (*None, int, tuple(int), optional*) −The dimension or dimensions to reduce. Defaults to None. If None, all dimensions are reduced.

Keyword Arguments

- **correction** (int, *optional*) −The difference between the sample size and sample degrees of freedom. Defaults to Bessel' s correction. Defaults to 1.
- **keepdim** (bool, *optional*) −Whether the output tensor has dim retained or not. If True , keep these reduced dimensions and the length is 1. If False, don't keep these dimensions. Defaults to False.

Returns

Tensor, the standard deviation. Suppose the shape of *input* is $(x_0, x_1, ..., x_R)$:

- If *dim* is () and *keepdim* is set to False , returns a 0-D Tensor, indicating the standard deviation of all elements in *input*.
- If *dim* is int, e.g. 1 and *keepdim* is set to False , then the returned Tensor has shape $(x_0, x_2, ..., x_R)$.
- If *dim* is tuple(int) or list(int), e.g. (1, 2) and *keepdim* is set to False , then the returned Tensor has shape $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** −If *input* is not a Tensor.
- **TypeError** −If *input* is not in bfloat16, float16, float32.
- **TypeError** −If *dim* is not one of the followings: None, int, tuple.
- **TypeError** −If *correction* is not an int.
- **TypeError** −If *keepdim* is not a bool.
- **ValueError** −If *dim* is out of range $[-input.ndim, input.ndim)$.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import mint, Tensor
>>> input = Tensor(np.array([[1, 2, 3], [-1, 1, 4]]).astype(np.float32))
>>> output = mint.std(input, dim=1, correction=1, keepdim=False)
>>> print(output)
[1.      2.5166113]
```

mindspore.mint.std_mean

mindspore.mint.**std_mean**(*input*, *dim=None*, *, *correction=1*, *keepdim=False*)

By default, return the standard deviation and mean of each dimension in Tensor. If dim is a dimension list, calculate the standard deviation and mean of the corresponding dimension.

The standard deviation (σ) is calculated as:

$$\sigma = \sqrt{\frac{1}{N-\delta N}\sum_{j=0}^{N-1}\left(self_{ij} - \overline{x_i}\right)^2}$$

where is x the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction* .

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor. Supported dtypes: float16, float32.
- **dim** (*Union[int, tuple(int), list(int)], optional*) –Specify the dimensions for calculating standard deviation and mean. Default value: None.

Keyword Arguments

- **correction** (int, *optional*) –Difference between the sample size and sample degrees of freedom. Defaults to Bessel's correction. Default: 1.
- **keepdim** (bool, *optional*) –Whether to preserve the dimensions of the output Tensor. If True, retain the reduced dimension with a size of 1. Otherwise, remove the dimensions. Default value: False.

Returns

A tuple of standard deviation and mean.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not one of the following data types: int, tuple, list, or Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> input = ms.Tensor([[1, 2, 3, 4], [-1, 1, 4, -10]], ms.float32)
>>> output_std, output_mean = ms.mint.std_mean(input, 1, correction=2, keepdim=True)
>>> print(output_std)
[[1.5811388]
 [7.3824115]]
>>> print(output_mean)
[[ 2.5]
 [-1.5]]
```

mindspore.mint.unique

mindspore.mint.**unique**(*input*, *sorted=True*, *return_inverse=False*, *return_counts=False*, *dim=None*)

Returns the unique elements of input tensor.

when *return_inverse=True*, also return a tensor containing the index of each value of input tensor corresponding to the output unique tensor. when *return_counts=True*, also return a tensor containing the number of occurrences for each unique value or tensor.

Parameters

- **input** (Tensor) –The input tensor.
- **sorted** (*bool, optional*) –Whether to sort the unique elements in ascending order before returning as output. Default: True .
- **return_inverse** (*bool, optional*) –Whether to also return the indices for where elements in the original input ended up in the returned unique list. Default: False .
- **return_counts** (*bool, optional*) –Whether to also return the counts for each unique element. Default: False .
- **dim** (*int, optional*) –the dimension to operate upon. If None, the unique of the flattened input is returned. Otherwise, each of the tensors indexed by the given dimension is treated as one of the elements to apply the unique operation upon. Default: None .

Returns

A tensor or a tuple of tensors containing some of tensor objects (*output*, *inverse_indices*, *counts*).

- output(Tensor) - The output tensor including the unique elements of input tensor, it has same dtype as input.
- inverse_indices(Tensor) - Return when return_inverse is True. It represents the indices for where elements in the original input map to in the output. When dim is None, it has same shape as input, otherwise, the shape is input.shape[dim].
- counts(Tensor) - Return when return_counts is True. It represents the number of occurrences for each unique value or tensor. When dim is None, it has same shape as output, otherwise, the shape is output.shape(dim).

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import mint
>>> x = Tensor(np.array([1, 2, 5, 2]), mindspore.int32)
>>> output = mint.unique(x, return_inverse=True)
>>> print(output)
(Tensor(shape=[3], dtype=Int32, value= [1, 2, 5]), Tensor(shape=[4], dtype=Int64, value= [0,␣
↪1, 2, 1]))
>>> y = output[0]
>>> print(y)
[1 2 5]
>>> idx = output[1]
>>> print(idx)
[0 1 2 1]
```

mindspore.mint.var

mindspore.mint.**var**(*input*, *dim=None*, *, *correction=1*, *keepdim=False*)

Calculates the variance over the dimensions specified by *dim*. *dim* can be a single dimension, list of dimensions, or None to reduce over all dimensions.

The variance (δ^2) is calculated as:

$$\delta^2 = \frac{1}{\max(0, N - \delta N)} \sum_{i=0}^{N-1} (x_i - \bar{x})^2$$

where x is the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The tensor used to calculate the variance.
- **dim** (*None, int, tuple(int), optional*) –The dimension or dimensions to reduce. Defaults to None. If None, all dimensions are reduced.

Keyword Arguments

- **correction** (int, *optional*) –The difference between the sample size and sample degrees of freedom. Defaults to Bessel' s correction. Defaults to 1.
- **keepdim** (bool, *optional*) –Whether the output tensor has dim retained or not. If True , keep these reduced dimensions and the length is 1. If False, don't keep these dimensions. Defaults to False.

Returns

Tensor, the variance. Suppose the shape of *input* is $(x_0, x_1, ..., x_R)$:

- If *dim* is () and *keepdim* is set to False , returns a 0-D Tensor, indicating the variance of all elements in *input*.
- If *dim* is int, e.g. 1 and *keepdim* is set to False , then the returned Tensor has shape $(x_0, x_2, ..., x_R)$.

- If *dim* is tuple(int) or list(int), e.g. `(1, 2)` and *keepdim* is set to `False` , then the returned Tensor has shape $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *input* is not in bfloat16, float16, flaot32.
- **TypeError** –If *dim* is not one of the followings: None, int, list, tuple.
- **TypeError** –If *correction* is not an int.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range $[-input.ndim, input.ndim)$.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[8, 2, 1], [5, 9, 3], [4, 6, 7]], mindspore.float32)
>>> output = mint.var(input, dim=0, correction=1, keepdim=True)
>>> print(output)
[[ 4.333333, 12.333333, 9.333333]]
```

mindspore.mint.var_mean

mindspore.mint.**var_mean**(*input*, *dim=None*, ***, *correction=1*, *keepdim=False*)

By default, return the variance and mean of each dimension in Tensor. If dim is a dimension list, calculate the variance and mean of the corresponding dimension.

The variance (σ^2) is calculated as:

$$\sigma^2 = \frac{1}{N - \delta N} \sum_{j=0}^{N-1} \left(self_{ij} - \overline{x_i}\right)^2$$

where is x the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the *correction* .

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor. Supported dtypes: float16, float32.
- **dim** (*Union[int, tuple(int), list(int)], optional*) –Specify the dimensions for calculating variance and mean. Default value: `None`.

Keyword Arguments

- **correction** (int, *optional*) –Difference between the sample size and sample degrees of freedom. Defaults to Bessel's correction. Default: `1`.

- **keepdim** (bool, optional) –Whether to preserve the dimensions of the output Tensor. If True, retain the reduced dimension with a size of 1. Otherwise, remove the dimensions. Default value: False.

Returns

A tuple of variance and mean.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *dim* is not one of the following data types: int, tuple, list, or Tensor.
- **TypeError** –If *keepdim* is not a bool.
- **ValueError** –If *dim* is out of range.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> input = ms.Tensor([[1, 2, 3, 4], [-1, 1, 4, -10]], ms.float32)
>>> output_var, output_mean = ms.mint.var_mean(input, 1, correction=2, keepdim=True)
>>> print(output_var)
[[ 2.5]
 [54.5]]
>>> print(output_mean)
[[ 2.5]
 [-1.5]]
```

4.3.3 Comparison Operations

API Name	Description	Supported Platforms
mindspore.mint.allclose	Returns a new Tensor with boolean elements representing if each element of *input* is "close" to the corresponding element of *other*.	Ascend
mindspore.mint.argsort	Sorts the input tensor along the given dimension in specified order and return the sorted indices.	Ascend
mindspore.mint.eq	Compute the equivalence of the two inputs element-wise.	Ascend
mindspore.mint.equal	Computes the equivalence between two tensors.	Ascend
mindspore.mint.greater	Compute the value of $input > other$ element-wise.	Ascend
mindspore.mint.greater_equal	Computes the boolean value of $input >= other$ element-wise.	Ascend
mindspore.mint.gt	Compute the value of $input > other$ element-wise.	Ascend
mindspore.mint.isclose	Return a boolean tensor where two tensors are element-wise equal within a tolerance.	Ascend
mindspore.mint.isfinite	Determine which elements are finite for each position.	Ascend

continues on next page

Table 6 – continued from previous page

mindspore.mint.isinf	Return a boolean tensor indicating which elements are +/- inifnity.	Ascend
mindspore.mint.isneginf	Determines which elements are -inf for each position.	Ascend
mindspore.mint.le	Compute the value of *input* <= *other* element-wise.	Ascend
mindspore.mint.less	Compute the value of *input* < *other* element-wise.	Ascend
mindspore.mint.less_equal	Compute the value of *input* <= *other* element-wise.	Ascend
mindspore.mint.lt	Alias for mindspore.mint.less() .	Ascend
mindspore.mint.maximum	Compute the maximum of the two input tensors element-wise.	Ascend
mindspore.mint.minimum	Compute the minimum of the two input tensors element-wise.	Ascend
mindspore.mint.ne	Compute the non-equivalence of two inputs element-wise.	Ascend
mindspore.mint.not_equal	Alias for mindspore.mint.ne() .	Ascend
mindspore.mint.topk	Finds values and indices of the *k* largest or smallest entries along a given dimension.	Ascend
mindspore.mint.sort	Sorts the elements of the input tensor along the given dimension in the specified order.	Ascend

mindspore.mint.allclose

mindspore.mint.**allclose**(*input*, *other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a new Tensor with boolean elements representing if each element of *input* is "close" to the corresponding element of *other*. Closeness is defined as:

$$|input - other| atol + rtol » |other|$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – First tensor to compare. Support dtype: float16, float32, float64, int8, int16, int32, int64 and uint8. On Ascend, more dtypes are support: bool and bfloat16.
- **other** (Tensor) – Second tensor to compare. Dtype must be same as *input*.
- **rtol** (*Union[float, int, bool], optional*) – Relative tolerance. Default: 1e-05 .
- **atol** (*Union[float, int, bool], optional*) – Absolute tolerance. Default: 1e-08 .
- **equal_nan** (*bool, optional*) – If True , then two NaNs will be considered equal. Default: False.

Returns

A bool Scalar.

Raises

- **TypeError** – *input* or *other* is not Tensor.

- **TypeError** – *input* or *other* dtype is not support.
- **TypeError** – *atol* or *rtol* is not float, int or bool.
- **TypeError** – *equal_nan* is not bool.
- **TypeError** – *input* and *other* have different dtypes.
- **ValueError** – *input* and *other* cannot broadcast.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1.3, 2.1, 3.2, 4.1, 5.1]), mindspore.float16)
>>> other = Tensor(np.array([1.3, 3.3, 2.3, 3.1, 5.1]), mindspore.float16)
>>> output = mint.allclose(input, other)
>>> print(output)
False
```

mindspore.mint.eq

mindspore.mint.**eq**(*input*, *other*)

Compute the equivalence of the two inputs element-wise.

$$out_i = \begin{cases} \text{True, if } input_i = other_i \\ \text{False, if } input_i \neq other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The input must be two Tensors, or a Tensor and a Scalar.
- The shapes of the inputs can be broadcasted to each other.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **other** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> x = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.eq(x, 2.0)
>>> print(output)
[False  True False]
>>> # case 2: The shape of two inputs are the same
>>> x = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> y = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.eq(x, y)
>>> print(output)
[ True  True False]
```

mindspore.mint.equal

mindspore.mint.**equal**(*input*, *other*)

Computes the equivalence between two tensors.

Note: *input* and *other* comply with the implicit type conversion rules to make the data types consistent.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The first input.
- **other** (Tensor) –The second input.

Returns

bool.

Raises

TypeError –If *input* or *other* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> x = Tensor([1, 2, 3], mindspore.int32)
>>> y = Tensor([1, 2, 4], mindspore.int32)
>>> output = mint.equal(x, y)
>>> print(output)
False
```

mindspore.mint.greater

mindspore.mint.**greater**(*input*, *other*)

Compute the value of $input > other$ element-wise.

Parameters

- **input** (*Union[Tensor, Number]*) – The first input.
- **other** (*Union[Tensor, Number]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.greater(input, 2.0)
>>> print(output)
[False False True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.greater(input, other)
>>> print(output)
[ False False False]
```

mindspore.mint.greater_equal

mindspore.mint.**greater_equal**(*input*, *other*) → *Tensor*

Computes the boolean value of $input >= other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i >= other_i \\ \text{False, if } input_i < other_i \end{cases}$$

Note:

- Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.
- The inputs must be two tensors or one tensor and one scalar.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them can be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If the input Tensor can be broadcast, the low dimension will be extended to the corresponding high dimension in another input by copying the value of the dimension.

Parameters

- **input** (*Union[Tensor, Number]*) –The first input is a number or a tensor whose data type is number or bool_.
- **other** (*Union[Tensor, Number]*) –Second input. When the first input is a Tensor, the second input should be a Number, or a Tensor of the number or **bool_** data type. When the first input is a Scalar, the second input must be a Tensor of number or **bool_** data type.

Returns

Tensor, the shape is the same as the one after broadcasting, and the data type is bool.

Raises

TypeError –If neither *input* nor *other* is a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3]), mindspore.int32)
>>> other = Tensor(np.array([1, 1, 4]), mindspore.int32)
>>> output = mint.greater_equal(input, other)
>>> print(output)
[True True False]
>>> y = 2.1
>>> output = mint.greater_equal(input, y)
>>> print(output)
[False False True]
```

mindspore.mint.gt

`mindspore.mint.gt(input, other)`

Compute the value of $input > other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i > other_i \\ \text{False, if } input_i <= other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The inputs must be two tensors or one tensor and one scalar.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them can be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If the input Tensor can be broadcast, the low dimension will be extended to the corresponding high dimension in another input by copying the value of the dimension.

Parameters

- **input** (*Union[Tensor, number.Number, bool]*) –The first input.
- **other** (*Union[Tensor, number.Number, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.gt(input, 2.0)
>>> print(output)
[False False True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.gt(input, other)
>>> print(output)
[ False False False]
```

mindspore.mint.isclose

`mindspore.mint.`**`isclose`**(*input*, *other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Return a boolean tensor where two tensors are element-wise equal within a tolerance. Math function is defined as:

$$|input - other| atol + rtol » |other|$$

Two Infinite values are considered equal if they have the same sign, Two NaN values are considered equal if *equal_nan* is `True` .

Parameters

- **input** (`Tensor`) –The first input tensor.
- **other** (`Tensor`) –The second input tensor.
- **rtol** (*Union[float, int, bool], optional*) –Relative tolerance. Default `1e-05` .
- **atol** (*Union[float, int, bool], optional*) –Absolute tolerance. Default `1e-08` .
- **equal_nan** (*bool, optional*) –Whether two NaNs are considered equal. Default `False` .

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> mindspore.mint.isclose(mindspore.tensor([2e6, float("inf"), float("-inf"), float("inf"),␣
↪float("nan")]),
...                         mindspore.tensor([2e7, float("inf"), float("-inf"), float("-inf"),␣
↪float("nan")]))
Tensor(shape=[5], dtype=Bool, value= [False,  True,  True, False, False])
>>>
>>> mindspore.mint.isclose(mindspore.tensor([1e6, 2e6, 3e6]),
...                         mindspore.tensor([1.00008e6, 2.00008e7, 3.00008e8]), rtol=1e3)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
>>>
>>> mindspore.mint.isclose(mindspore.tensor([1e6, 2e6, 3e6]),
...                         mindspore.tensor([1.00001e6, 2.00002e6, 3.00009e6]), atol=1e3)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
>>> mindspore.mint.isclose(mindspore.tensor([float("nan"), 1, 2]),
...                         mindspore.tensor([float("nan"), 1, 2]), equal_nan=True)
Tensor(shape=[3], dtype=Bool, value= [ True,  True,  True])
```

mindspore.mint.isfinite

mindspore.mint.**isfinite**(*input*)

Determine which elements are finite for each position. If elements are not NaN , -INF , INF, they are finite.

$$out_i = \begin{cases} \text{if } input_i = \text{Finite}, & True \\ \text{if } input_i \neq \text{Finite}, & False \end{cases}$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor, has the same shape of input, and the dtype is bool.

Raises

TypeError – If input is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([np.log(-1), 1, np.log(0)]), mindspore.float32)
>>> output = mint.isfinite(x)
>>> print(output)
[False True False]
>>> x = Tensor(2.1, mindspore.float64)
>>> output = mint.isfinite(x)
>>> print(output)
True
```

mindspore.mint.isinf

mindspore.mint.**isinf**(*input*)

Return a boolean tensor indicating which elements are +/- inifnity.

Warning:

- This is an experimental API that is subject to change.
- For Ascend, it is only supported on platforms above Atlas A2.

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([-1, 3, float("inf"), float("-inf"), float("nan")])
>>> mindspore.mint.isinf(input)
Tensor(shape=[5], dtype=Bool, value= [False, False,  True,  True, False])
```

mindspore.mint.isneginf

mindspore.mint.**isneginf**(*input*)

Determines which elements are -inf for each position.

Warning:

- This API can be used only on the Atlas A2 training series.

Parameters

input (Tensor) – Input Tensor.

Returns

Tensor with the same shape as the input, where elements are *True* if the corresponding element in the *input* is negative infinity, and *False* otherwise.

Raises

TypeError – If the input is not a tensor.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint, Tensor
>>> from mindspore import dtype as mstype
>>> output = mint.isneginf(Tensor([[-float("inf"), float("inf")], [1, -float("inf")]],␣
↪mstype.float32))
>>> print(output)
[[ True False]
 [False  True]]
```

mindspore.mint.le

mindspore.mint.**le**(*input*, *other*)

Compute the value of $input <= other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i <= other_i \\ \text{False, if } input_i > other_i \end{cases}$$

Note:

- Support implicit type conversion.
- The inputs must be two tensors or one tensor and one scalar.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.le(input, 2.0)
>>> print(output)
[True  True False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.le(input, other)
>>> print(output)
[ True  True  True]
```

mindspore.mint.less

mindspore.mint.**less**(*input*, *other*)

Compute the value of $input < other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i < other_i \\ \text{False, if } input_i >= other_i \end{cases}$$

Note: Support implicit type conversion.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.less(input, 2.0)
>>> print(output)
[True  False False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.less(input, other)
>>> print(output)
[ False  False  True]
```

mindspore.mint.less_equal

mindspore.mint.**less_equal**(*input*, *other*)

Compute the value of $input <= other$ element-wise.

$$out_i = \begin{cases} \text{True, if } input_i <= other_i \\ \text{False, if } input_i > other_i \end{cases}$$

Note:

- Support implicit type conversion.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.less_equal(input, 2.0)
>>> print(output)
[True  True False]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.less_equal(input, other)
>>> print(output)
[ True  True  True]
```

mindspore.mint.lt

mindspore.mint.**lt**(*input*, *other*)

Alias for *mindspore.mint.less()*.

Supported Platforms:

`Ascend`

mindspore.mint.maximum

mindspore.mint.**maximum**(*input*, *other*)

Compute the maximum of the two input tensors element-wise.

$$output_i = \max(input_i, other_i)$$

Note:

- Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.
- If one of the elements being compared is a NaN, then that element is returned.

Warning: If all inputs are scalar of integers. In Graph mode, the output will be Tensor of int32, while in PyNative mode, the output will be Tensor of int64.

Parameters

- **input** (*Union[Tensor, Number, bool]*) – The first input.
- **other** (*Union[Tensor, Number, bool]*) – The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> # case 1 : same data type
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.float32)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float32)
>>> mindspore.mint.maximum(input, other)
Tensor(shape=[3], dtype=Float32, value= [ 4.00000000e+00,  5.00000000e+00,  6.00000000e+00])
>>>
>>> # case 2 : the data type is the one with higher precision or higher digits among the two␣
↪inputs.
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.int64)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float64)
>>> mindspore.mint.maximum(input, other)
Tensor(shape=[3], dtype=Float64, value= [ 4.00000000e+00,  5.00000000e+00,  6.00000000e+00])
```

mindspore.mint.minimum

`mindspore.mint.`**`minimum`**(*input*, *other*)

Compute the minimum of the two input tensors element-wise.

$$output_i = \min(input_i, other_i)$$

Note:

- Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.
- When the inputs are two tensors, dtypes of them cannot be bool at the same time.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Shapes of them are supposed to be broadcast.
- If one of the elements being compared is a NaN, then that element is returned.

Parameters

- **input** (*Union[Tensor, Number, bool]*) –The first input.
- **other** (*Union[Tensor, Number, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> # case 1 : same data type
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.float32)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float32)
>>> mindspore.mint.minimum(input, other)
Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00])
>>>
>>> # case 2 : the data type is the one with higher precision or higher digits among the two␣
 ↪inputs.
>>> input = mindspore.tensor([1.0, 5.0, 3.0], mindspore.int64)
>>> other = mindspore.tensor([4.0, 2.0, 6.0], mindspore.float64)
>>> mindspore.mint.minimum(input, other)
Tensor(shape=[3], dtype=Float64, value= [ 1.00000000e+00,  2.00000000e+00,  3.00000000e+00])
```

mindspore.mint.ne

`mindspore.mint.ne`(*input*, *other*)

Compute the non-equivalence of two inputs element-wise.

$$out_i = \begin{cases} \text{True, if } input_i \neq other_i \\ \text{False, if } input_i = other_i \end{cases}$$

Note:

- Support implicit type conversion.
- When the inputs are two tensors, the shapes of them could be broadcast.
- When the inputs are one tensor and one scalar, the scalar could only be a constant.
- Broadcasting is supported.

Parameters

- **input** (*Union[Tensor, Number, bool]*) –The first input.
- **other** (*Union[Tensor, Number, bool]*) –The second input.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> # case 1: The shape of two inputs are different
>>> input = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.ne(input, 2.0)
>>> print(output)
[True False  True]
>>> # case 2: The shape of two inputs are the same
>>> input = mindspore.tensor([1, 2, 3], mindspore.int32)
>>> other = mindspore.tensor([1, 2, 4], mindspore.int32)
>>> output = mindspore.mint.ne(input, other)
>>> print(output)
[ False False  True]
```

mindspore.mint.not_equal

mindspore.mint.**not_equal**(*input*, *other*)

Alias for *mindspore.mint.ne()* .

Supported Platforms:

`Ascend`

mindspore.mint.topk

mindspore.mint.**topk**(*input*, *k*, *dim=- 1*, *largest=True*, *sorted=True*)

Finds values and indices of the *k* largest or smallest entries along a given dimension.

Warning:

- If sorted is set to False, due to different memory layout and traversal methods on different platforms, the display order of calculation results may be inconsistent when *sorted* is False.

If the *input* is a one-dimensional Tensor, finds the *k* largest or smallest entries in the Tensor, and outputs its value and index as a Tensor. values[*k*] is the *k* largest item in *input*, and its index is indices [*k*].

For a multi-dimensional matrix, calculates the first or last *k* entries in a given dimension, therefore:

$$values.shape = indices.shape$$

If the two compared elements are the same, the one with the smaller index value is returned first.

Parameters

- **input** (`Tensor`) – Input to be computed.
- **k** (*int*) – The number of top or bottom elements to be computed along the last dimension.
- **dim** (*int, optional*) – The dimension to sort along. Default: `-1` .
- **largest** (*bool, optional*) – If largest is `False` then the k smallest elements are returned. Default: `True` .

- **sorted** (*bool, optional*) – If `True` , the obtained elements will be sorted by the values in descending order or ascending order according to *largest*. If `False` , the obtained elements will not be sorted. Default: `True` .

Returns

A tuple consisting of *values* and *indices*.

- values (Tensor) - The *k* largest or smallest elements in each slice of the given dimension.
- indices (Tensor) - The indices of values within the last dimension of input.

Raises

- **TypeError** – If *sorted* is not a bool.
- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *k* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> x = ms.Tensor([[0.5368, 0.2447, 0.4302, 0.9673],
...                [0.4388, 0.6525, 0.4685, 0.1868],
...                [0.3563, 0.5152, 0.9675, 0.8230]], dtype=ms.float32)
>>> output = mint.topk(x, 2, dim=1)
>>> print(output)
(Tensor(shape=[3, 2], dtype=Float32, value=
[[ 9.67299998e-01,  5.36800027e-01],
 [ 6.52499974e-01,  4.68499988e-01],
 [ 9.67499971e-01,  8.23000014e-01]]), Tensor(shape=[3, 2], dtype=Int64, value=
[[3, 0],
 [1, 2],
 [2, 3]]))
>>> output2 = mint.topk(x, 2, dim=1, largest=False)
>>> print(output2)
(Tensor(shape=[3, 2], dtype=Float32, value=
[[ 2.44700000e-01,  4.30200011e-01],
 [ 1.86800003e-01,  4.38800007e-01],
 [ 3.56299996e-01,  5.15200019e-01]]), Tensor(shape=[3, 2], dtype=Int64, value=
[[1, 2],
 [3, 0],
 [0, 1]]))
```

mindspore.mint.sort

mindspore.mint.**sort**(*input*, *, *dim=- 1*, *descending=False*, *stable=False*)

Sorts the elements of the input tensor along the given dimension in the specified order.

> **Warning:** Currently, the data types of float16, uint8, int8, int16, int32, int64 are well supported. If use float32, it may cause loss of accuracy.

Parameters

input (Tensor) – The input tensor to sort. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Keyword Arguments

- **dim** (int, *optional*) – The dimension to sort along. Default: -1, means the last dimension.
- **descending** (bool, *optional*) – Controls the sort order. If *descending* is True, the elements are sorted in descending order, or else sorted in ascending order. Default: False .
- **stable** (bool, *optional*) – Controls the sort order. If stable is True then the sorting routine becomes stable, preserving the order of equivalent elements. Default: False .

Returns

- y1, a tensor whose values are the sorted values, with the same shape and data type as input.
- y2, a tensor that consists of the indices of the elements in the original input tensor. Data type is int64.

Raises

- **TypeError** – If *dim* is not an int.
- **TypeError** – If *descending* is not a bool.
- **TypeError** – If *input* not in float16, float32, uint8, int8, int16, int32, int64, bfloat16
- **TypeError** – If *stable* is not a bool.
- **ValueError** – If *dim* is not in range of [-len(input.shape), len(input.shape)).

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> output = mint.sort(x)
>>> # The output below is based on the Ascend platform.
>>> print(output)
(Tensor(shape=[3, 3], dtype=Float16, value=
[[ 1.0000e+00,  2.0000e+00,  8.0000e+00],
[ 3.0000e+00,  5.0000e+00,  9.0000e+00],
[ 4.0000e+00,  6.0000e+00,  7.0000e+00]]), Tensor(shape=[3, 3], dtype=Int64, value=
[[2, 1, 0],
[2, 0, 1],
[0, 1, 2]]))
```

4.3.4 BLAS and LAPACK Operations

API Name	Description	Supported Platforms
mindspore.mint.addbmm	Applies batch matrix multiplication to *batch1* and *batch2*, with a reduced add step and add *input* to the result.	Ascend
mindspore.mint.addmm	Performs a matrix multiplication of the 2-D matrices mat1 and mat2.	Ascend
mindspore.mint.baddbmm	The result is the sum of the input and a batch matrix-matrix product of matrices in batch1 and batch2.	Ascend
mindspore.mint.bmm	Performs batch matrix-matrix multiplication of two three-dimensional tensors.	Ascend
mindspore.mint.dot	Computes the dot product of two 1D tensor.	Ascend
mindspore.mint.inverse	Compute the inverse of the input matrix.	Ascend
mindspore.mint.matmul	Return the matrix product of two tensors.	Ascend
mindspore.mint.meshgrid	Generates coordinate matrices from given coordinate tensors.	Ascend
mindspore.mint.mm	Returns the matrix product of two arrays.	Ascend
mindspore.mint.outer	Return outer product of *input* and *vec2*.	Ascend
mindspore.mint.trace	Returns a new tensor that is the sum of the *input* main trace.	Ascend

mindspore.mint.addbmm

mindspore.mint.**addbmm**(*input*, *batch1*, *batch2*, *, *beta=1*, *alpha=1*)

Applies batch matrix multiplication to *batch1* and *batch2*, with a reduced add step and add *input* to the result.

The optional value *alpha* is the matrix-matrix product between *batch1* and *batch2*, and *beta* is the scale factor for the added tensor *input*. If *beta* is 0, then *input* will be ignored.

$$output = \beta input + \alpha(\sum_{i=0}^{b-1} batch1_i @ batch2_i)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Tensor to be added.
- **batch1** (Tensor) –The first batch of tensor to be multiplied.
- **batch2** (Tensor) –The second batch of tensor to be multiplied.

Keyword Arguments

- **beta** (*Union[int, float], optional*) –Multiplier for *input*. Default: 1 .
- **alpha** (*Union[int, float], optional*) –Multiplier for *batch1* @ *batch2*. Default: 1 .

Returns

Tensor, has the same dtype as *input*.

Raises

- **TypeError** –If *alpha* or *beta* is not an int or float.
- **ValueError** –If *batch1*, *batch2* cannot apply batch matrix multiplication.
- **ValueError** –If *batch1* and *batch2* are not 3-D tensors.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> m = np.ones((3, 3)).astype(np.float32)
>>> arr1 = np.arange(24).astype(np.float32).reshape((2, 3, 4))
>>> arr2 = np.arange(24).astype(np.float32).reshape((2, 4, 3))
>>> a = Tensor(arr1)
>>> b = Tensor(arr2)
>>> c = Tensor(m)
>>> output = mint.addbmm(c, a, b)
>>> print(output)
[[ 949. 1009. 1069.]
 [1285. 1377. 1469.]
 [1621. 1745. 1869.]]
```

mindspore.mint.addmm

mindspore.mint.**addmm**(*input*, *mat1*, *mat2*, *, *beta=1*, *alpha=1*)

Performs a matrix multiplication of the 2-D matrices mat1 and mat2. The matrix input is added to the final result. The formula is defined as follows:

$$output = \beta input + \alpha(mat1@mat2)$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –matrix to be added, the shape must be broadcastable with $mat1@mat2$.
- **mat1** (Tensor) –the first matrix to be matrix multiplied, must be 2-D Tensor, with the same shape of the input.
- **mat2** (Tensor) –the second matrix to be matrix multiplied, must be 2-D Tensor, with the same shape of the input.

Keyword Arguments

- **beta** (*Union[float, int], optional*) –multiplier for input. Default: `1` .
- **alpha** (*Union[float, int], optional*) –multiplier for $mat1@mat2$. Default: `1` .

Returns

Tensor, with the same dtype as *input* and the same shape as $mat1@mat2$.

Raises

- **TypeError** –If the type of *input*, *mat1* or *mat2* is not Tensor.
- **TypeError** –If the types of *input*, *mat1*, *mat2* are different.
- **ValueError** –If *mat1* and *mat2* are not 2-D tensors.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones([3, 3]).astype(np.float32))
>>> mat1 = Tensor(np.ones([3, 4]).astype(np.float32))
>>> mat2 = Tensor(np.ones([4, 3]).astype(np.float32))
>>> output =  mint.addmm(input, mat1, mat2)
>>> print(output)
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

mindspore.mint.baddbmm

mindspore.mint.**baddbmm**(*input*, *batch1*, *batch2*, *, *beta=1*, *alpha=1*)

The result is the sum of the input and a batch matrix-matrix product of matrices in batch1 and batch2. The formula is defined as follows:

$$\text{out}_i = \beta\text{input}_i + \alpha(\text{batch1}_i \mathbin{@} \text{batch2}_i)$$

Parameters

- **input** (Tensor) –The input Tensor. When batch1 is a (C, W, T) Tensor and batch2 is a (C, T, H) Tensor, input must be broadcastable with (C, W, H) Tensor.
- **batch1** (Tensor) –$batch1$ in the above formula. Must be 3-D Tensor, dtype is same as input.
- **batch2** (Tensor) –$batch2$ in the above formula. Must be 3-D Tensor, dtype is same as input.

Keyword Arguments

- **beta** (*Union[float, int], optional*) –multiplier for input. Default: `1` .
- **alpha** (*Union[float, int], optional*) –multiplier for $batch1@batch2$. Default: `1` .

Returns

Tensor, has the same dtype as input, shape will be (C, W, H).

Raises

- **TypeError** –If the type of *input*, *batch1*, *batch2* is not Tensor.
- **TypeError** –If the types of *input*, *batch1*, *batch2* are different.
- **ValueError** –If *batch1* and *batch2* are not 3-D tensors.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones([1, 3, 3]).astype(np.float32))
>>> batch1 = Tensor(np.ones([1, 3, 4]).astype(np.float32))
>>> batch2 = Tensor(np.ones([1, 4, 3]).astype(np.float32))
>>> output = mint.baddbmm(input, batch1, batch2)
>>> print(output)
[[[5. 5. 5.]
  [5. 5. 5.]
  [5. 5. 5.]]]
```

mindspore.mint.bmm

mindspore.mint.**bmm**(*input*, *mat2*)

Performs batch matrix-matrix multiplication of two three-dimensional tensors.

$$\text{output} = \text{input}@\text{mat2}$$

Parameters

- **input** (Tensor) – The first batch of matrices to be multiplied. Must be a three-dimensional tensor of shape *(b, n, m)*.
- **mat2** (Tensor) – The second batch of matrices to be multiplied. Must be a three-dimensional tensor of shape *(b, m, p)*.

Returns

Tensor, the output tensor of shape *(b, n, p)*, where each matrix is the product of the corresponding matrices in the input batches.

Raises

- **ValueError** – If *input* or *mat2* is not three-dimensional tensors.
- **ValueError** – If the length of the third dimension of *input* is not equal to the length of the second dimension of *mat2*.
- **ValueError** – If the batch size of the inputs is not equal to the batch size of the mat2.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> a = Tensor(np.ones(shape=[2, 3, 4]), mindspore.float32)
>>> b = Tensor(np.ones(shape=[2, 4, 5]), mindspore.float32)
>>> output = mint.bmm(a, b)
>>> print(output)
[[[4. 4. 4. 4. 4.]
  [4. 4. 4. 4. 4.]
  [4. 4. 4. 4. 4.]]
 [[4. 4. 4. 4. 4.]
```

(continues on next page)

(continued from previous page)

```
  [4. 4. 4. 4. 4.]
  [4. 4. 4. 4. 4.]]]
```

mindspore.mint.dot

mindspore.mint.**dot**(*input*, *other*)

Computes the dot product of two 1D tensor.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The first input in the dot product, must be 1D.
- **other** (Tensor) –The second input in the dot product, must be 1D.

Returns

Tensor, the shape is [] and the data type is same as *input*.

Raises

- **TypeError** –If dtype of *input*, *other* is not tensor.
- **TypeError** –If dtype of *input*, *other* are not in float16, float32 or bfloat16.
- **RuntimeError** –If dtypes of *input* and *other* are not same.
- **RuntimeError** –If shapes of *input* and *other* are not same.
- **RuntimeError** –If shapes of *input* and *other* are not 1D.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> x = Tensor([2.0, 3.0], mindspore.float32)
>>> y = Tensor([2.0, 1.0], mindspore.float32)
>>> output = mint.dot(x, y)
>>> print(output)
7.0
>>> print(output.dtype)
Float32
```

mindspore.mint.inverse

mindspore.mint.**inverse**(*input*)

Compute the inverse of the input matrix.

Parameters

input (Tensor) – A matrix to be calculated. Input *input* must be at least two dimensions, and the size of the last two dimensions must be the same size. And the matrix must be invertible.

Returns

Tensor, has the same type and shape as input *input*.

Raises

- **ValueError** – If the size of the last two dimensions of *input* is not the same.
- **ValueError** – If *input* is not empty and its dimensions are less than 2.
- **ValueError** – If the dimensions of *input* are larger than 6.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor([[1., 2.], [3., 4.]], mstype.float32)
>>> print(mint.inverse(x))
[[-2.   1. ]
 [ 1.5 -0.5]]
```

mindspore.mint.matmul

mindspore.mint.**matmul**(*input*, *other*)

Return the matrix product of two tensors.

Note:

- The dtype of *input* and *other* must be same.
- On Ascend, the rank of *input* or *other* must be between 1 and 6.

Parameters

- **input** (Tensor) – The first input tensor.
- **other** (Tensor) – The second input tensor.

Returns

Tensor or scalar

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # case 1 : Reasonable application of broadcast mechanism.
>>> input = mindspore.ops.arange(24, dtype=mindspore.float32).reshape(2, 3, 4)
>>> other = mindspore.ops.arange(20, dtype=mindspore.float32).reshape(4, 5)
>>> output = mindspore.mint.matmul(input, other)
>>> print(output)
[[[  70,   76,   82,   88,   94],
  [ 190,  212,  234,  256,  278],
  [ 310,  348,  386,  424,  462]],
 [[ 430,  484,  538,  592,  646],
  [ 550,  620,  690,  760,  830],
  [ 670,  756,  842,  928, 1014]]]
>>>
>>> # case 2 : The rank of `input` is 1.
>>> input = mindspore.ops.ones(([1, 2]))
>>> other = mindspore.ops.ones(([2]))
>>> mindspore.mint.matmul(input, other)
Tensor(shape=[1], dtype=Float32, value= [ 2.00000000e+00])
```

mindspore.mint.meshgrid

mindspore.mint.**meshgrid**(**tensors*, *indexing='ij'*)

Generates coordinate matrices from given coordinate tensors.

Given N one-dimensional coordinate tensors, returns a tuple outputs of N N-D coordinate tensors for evaluating expressions on an N-D grid.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

tensors (*Union(tuple[Tensor], list[Tensor])*) – In GRAPH_MODE, a tuple of N 1-D Tensor objects and the length of input should be greater than 1. In PYNATIVE_MODE, a tuple of N 0-D or 1-D Tensor objects and the length of input should be greater than 0. The data type is Number.

Keyword Arguments

indexing (*str, optional*) – Cartesian ('xy', default) or matrix ('ij') indexing of output. Valid options: xy' or `'ij'`. In the 2-D case with inputs of length M and N, for `'xy'` indexing, the shape of outputs is (N, M) for `'ij'` indexing, the shape of outputs is (M, N). In the 3-D case with inputs of length M, N and P, for `'xy'` indexing, the shape of outputs is (N, M, P) and for `'ij'` indexing, the shape of outputs is (M, N, P). Default: `'ij'` .

Returns

Tensors, a Tuple of N N-D Tensor objects. The data type is the same with the Inputs.

Raises

- **TypeError** – If *indexing* is not a str or *tensors* is not a tuple.
- **ValueError** – If *indexing* is neither `'xy'` nor `'ij'`.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([1, 2, 3, 4]).astype(np.int32))
>>> y = Tensor(np.array([5, 6, 7]).astype(np.int32))
>>> z = Tensor(np.array([8, 9, 0, 1, 2]).astype(np.int32))
>>> output = mint.meshgrid(x, y, z, indexing='xy')
>>> print(output)
(Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
   [2, 2, 2, 2, 2],
   [3, 3, 3, 3, 3],
   [4, 4, 4, 4, 4]]]),
 Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5],
   [5, 5, 5, 5, 5]],
  [[6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6],
   [6, 6, 6, 6, 6]],
  [[7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7],
   [7, 7, 7, 7, 7]]]),
 Tensor(shape=[3, 4, 5], dtype=Int32, value=
 [[[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2],
   [8, 9, 0, 1, 2]]]))
```

mindspore.mint.mm

mindspore.mint.**mm**(*input*, *mat2*)

Returns the matrix product of two arrays. If *input* is a $(n \times m)$ Tensor, *mat2* is a $(m \times p)$ Tensor, *out* will be a $(n \times p)$ Tensor.

Note: This function cannot support broadcasting. Refer to *mindspore.ops.matmul()* instead if you need a broadcastable function.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The first matrix of matrix multiplication. The last dimension of *input* must be the same size as the first dimension of *mat2*.
- **mat2** (Tensor) –The second matrix of matrix multiplication. The last dimension of *input* must be the same size as the first dimension of *mat2*.

Returns

Tensor, the matrix product of the inputs.

Raises

- **ValueError** –If the last dimension of *input* is not the same size as the second-to-last dimension of *mat2*.
- **TypeError** –If *input* or *mat2* is not a Tensor.
- **TypeError** –If dtype of *input* or *mat2* is not float16, float32 or bfloat16.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> x1 = ms.Tensor(np.random.rand(2, 3), ms.float32)
>>> x2 = ms.Tensor(np.random.rand(3, 4), ms.float32)
>>> out = mint.mm(x1, x2)
>>> print(out.shape)
(2, 4)
```

mindspore.mint.outer

mindspore.mint.**outer**(*input*, *vec2*)

Return outer product of *input* and *vec2*. If *input* is a vector of size n and *vec2* is a vector of size m , then output must be a matrix of shape (n, m) .

> **Warning:** This is an experimental API that is subject to change or deletion.

Note: This function does not broadcast.

Parameters

- **input** (Tensor) – 1-D input vector.
- **vec2** (Tensor) – 1-D input vector.

Returns

out, 2-D matrix, the outer product of two vectors.

Raises

- **TypeError** – If *input* or *vec2* is not a Tensor.
- **TypeError** – The implicitly converted data types of *input* and *vec2* are not one of float16, float32, float64, bool, uint8, int8, int16, int32, int64, complex64, complex128, bfloat16
- **ValueError** – If the dimension of *input* or *vec2* is not equal to 1.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> input = Tensor(np.array([7, 8, 9]), mindspore.int32)
>>> vec2 = Tensor(np.array([7, 10, 11]), mindspore.int32)
>>> out = mint.outer(input, vec2)
>>> print(out)
[[49 70 77]
 [56 80 88]
 [63 90 99]]
```

mindspore.mint.trace

mindspore.mint.**trace**(*input*)

Returns a new tensor that is the sum of the *input* main trace.

Parameters

input (Tensor) –2-D Tensor.

Returns

Tensor, when the data type of *input* is integer or bool, its data type is int64, otherwise it is the same as *input*, and size equals to 1.

Raises

- **TypeError** –If *input* is not a Tensor.
- **ValueError** –If the dimension of *input* is not equal to 2.
- **TypeError** –If the dtype of *input* is not one of float16, float32, float64, bool, uint8, int8, int16, int32, int64, complex64, complex128, bfloat16.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[10, 11, 12], [13, 14, 15], [16, 17, 18]]), mindspore.float32)
>>> output = mint.trace(input)
>>> print(output)
42.0
>>> input = Tensor(np.arange(1, 13).reshape(3, 4), mindspore.float32)
>>> output = mint.trace(input)
>>> print(output)
18.0
>>> input = Tensor(np.arange(12, 0, -1).reshape(4, 3), mindspore.float32)
>>> output = mint.trace(input)
>>> print(output)
24.0
```

4.3.5 Other Operations

API Name	Description	Supported Platforms
mindspore.mint.broadcast_to	Broadcasts input tensor to a given shape.	Ascend
mindspore.mint.cdist	Computes p-norm distance between each pair of row vectors of two input Tensors.	Ascend
mindspore.mint.cummax	Returns a tuple (values, indices) where *values* is the cumulative maximum value of input Tensor *input* along the dimension *dim*, and *indices* is the index location of each maximum value.	Ascend

continues on next page

Table 8 – continued from previous page

mindspore.mint.cummin	Returns a tuple (values, indices) where *values* is the cumulative minimum value of input Tensor *input* along the dimension *dim*, and *indices* is the index location of each minimum value.	Ascend
mindspore.mint.cumsum	Computes the cumulative sum of input Tensor along *dim*.	Ascend
mindspore.mint.diag	If input is a vector (1-D tensor), then returns a 2-D square tensor with the elements of input as the diagonal.	Ascend
mindspore.mint.flatten	Flatten a tensor along dimensions from *start_dim* to *end_dim*.	Ascend
mindspore.mint.flip	Reverses elements in a tensor along the given dims.	Ascend
mindspore.mint.repeat_interleave	Repeat elements of a tensor along an axis, like mindspore.numpy.repeat().	Ascend
mindspore.mint.searchsorted	Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.	Ascend
mindspore.mint.tril	Zero the input tensor above the diagonal specified.	Ascend
mindspore.mint.triangular_solve	Solves a system of equations with a square upper or lower triangular invertible matrix *A* and multiple right-hand sides *b*.	Ascend

mindspore.mint.broadcast_to

mindspore.mint.**broadcast_to**(*input*, *shape*)

Broadcasts input tensor to a given shape. The dim of input must be smaller than or equal to that of target. Suppose input shape is $(x_1, x_2, ..., x_m)$, target shape is $(*, y_1, y_2, ..., y_m)$, where $*$ means any additional dimension. The broadcast rules are as follows:

Compare the value of x_m and y_m, x_{m-1} and y_{m-1}, $\cdots$, x_1 and y_1 consecutively and decide whether these shapes are broadcastable and what the broadcast result is.

If the value pairs at a specific dim are equal, then that value goes right into that dim of output shape. With an input shape $(2, 3)$, target shape $(2, 3)$, the inferred output shape is $(2, 3)$.

If the value pairs are unequal, there are three cases:

Case 1: If the value of the target shape in the dimension is -1, the value of the output shape in the dimension is the value of the corresponding input shape in the dimension. With an input shape $(3, 3)$, target shape $(-1, 3)$, the output shape is $(3, 3)$.

Case 2: If the value of target shape in the dimension is not -1, but the corresponding value in the input shape is 1, then the corresponding value of the output shape is that of the target shape. With an input shape $(1, 3)$, target shape $(8, 3)$, the output shape is $(8, 3)$.

Case 3: If the corresponding values of the two shapes do not satisfy the above cases, it means that broadcasting from the input shape to the target shape is not supported.

So far we got the last m dims of the outshape, now focus on the first $*$ dims, there are two cases:

If the first $*$ dims of output shape does not have -1 in it, then fill the input shape with ones until their length are the same, and then refer to Case 2 mentioned above to calculate the output shape. With target shape $(3, 1, 4, 1, 5, 9)$, input shape $(1, 5, 9)$, the filled input shape will be $(1, 1, 1, 1, 5, 9)$ and thus the output shape is $(3, 1, 4, 1, 5, 9)$.

If the first $*$ dims of output shape have -1 in it, it implies this -1 is corresponding to a non-existing dim so they're not broadcastable. With target shape $(3, -1, 4, 1, 5, 9)$, input shape $(1, 5, 9)$, instead of operating the dim-filling process first, it raises errors directly.

Parameters

- **input** (Tensor) –The input tensor.
- **shape** (*tuple*) –The target shape.

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> shape = (2, 3)
>>> x = mindspore.tensor([1, 2, 3], mindspore.float32)
>>> output = mindspore.mint.broadcast_to(x, shape)
>>> print(output)
[[1. 2. 3.]
 [1. 2. 3.]]
>>> shape = (-1, 2)
>>> x = mindspore.tensor([[1], [2]], mindspore.float32)
>>> output = mindspore.mint.broadcast_to(x, shape)
>>> print(output)
[[1. 1.]
 [2. 2.]]
```

mindspore.mint.cdist

mindspore.mint.**cdist**(*x1*, *x2*, *p=2.0*, *compute_mode='use_mm_for_euclid_dist_if_necessary'*)

Computes p-norm distance between each pair of row vectors of two input Tensors.

Warning: This is an experimental optimizer API that is subject to change.

Note: On Ascend, the supported dtypes are float16 and float32.

Parameters

- **x1** (Tensor) –Input tensor of shape (B, P, M). Letter B represents 0 or positive int number. When B is equal to 0, it means this dimension can be ignored, i.e. shape of the tensor is (P, M).
- **x2** (Tensor) –Input tensor of shape (B, R, M), has the same dtype as *x1*.
- **p** (*float, optional*) –P value for the p-norm distance to calculate between each vector pair, P >= 0. Default: `2.0`.
- **compute_mode** (*string, optional*) –Specify the cumpute mode. Setting this parameter currently has no effect. Default: `'use_mm_for_euclid_dist_if_necessary'`.

Returns

Tensor, p-norm distance, has the same dtype as *x1*, its shape is (B, P, R).

Raises

- **TypeError** –If *x1* or *x2* is not Tensor.
- **TypeError** –If dtype of *x1* or *x2* is not listed in the "Note" above.
- **TypeError** –If *p* is not float32.
- **ValueError** –If *p* is negative.
- **ValueError** –If dimension of *x1* is not the same as *x2*.
- **ValueError** –If dimension of *x1* or *x2* is neither 2 nor 3.
- **ValueError** –If the batch dim of *x1* and *x2* can not broadcast.
- **ValueError** –If the number of columns of *x1* is not the same as that of *x2*.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[[1.0, 1.0], [2.0, 2.0]]]).astype(np.float32))
>>> y = Tensor(np.array([[[3.0, 3.0], [3.0, 3.0]]]).astype(np.float32))
>>> output = mint.cdist(x, y, 2.0)
>>> print(output)
[[[2.8284273 2.8284273]
  [1.4142137 1.4142137]]]
```

mindspore.mint.cummax

mindspore.mint.**cummax**(*input*, *dim*)

Returns a tuple (values, indices) where *values* is the cumulative maximum value of input Tensor *input* along the dimension *dim*, and *indices* is the index location of each maximum value.

$$y_i = \max(x_1, x_2, ..., x_i)$$

Note: O2 mode is not supported in Ascend.

Parameters

- **input** (Tensor) –The input Tensor. Rank of *input* must be greater than 0.
- **dim** (*int*) –The dimension to do the operation over. The value of *dim* must be in the range *[-input.ndim, input.ndim - 1]*.

Returns

tuple [Tensor], tuple of 2 Tensors, containing the cumulative maximum of elements and the index. The shape of each output tensor is the same as that of input *input*.

Raises

- **TypeError** –If *input* is not a Tensor.

- **TypeError** –If *dim* is not an int.
- **ValueError** –If *dim* is out the range of *[-input.ndim, input.ndim - 1]*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([[3, 4, 6, 10], [1, 6, 7, 9], [4, 3, 8, 7], [1, 3, 7, 9]]).astype(np.
↪float32))
>>> output = mint.cummax(x, dim=0)
>>> print(output[0])
[[ 3.  4.  6. 10.]
 [ 3.  6.  7. 10.]
 [ 4.  6.  8. 10.]
 [ 4.  6.  8. 10.]]
>>> print(output[1])
[[0 0 0 0]
 [0 1 1 0]
 [2 1 2 0]
 [2 1 2 0]]
```

mindspore.mint.cummin

`mindspore.mint.`**`cummin`**(*input*, *dim*)

Returns a tuple (values, indices) where *values* is the cumulative minimum value of input Tensor *input* along the dimension *dim*, and *indices* is the index location of each minimum value.

$$y_i = \min(x_1, x_2, ..., x_i)$$

Note: O2 mode is not supported in Ascend.

Parameters

- **input** (`Tensor`) –The input Tensor, The dimension must be greater than 0.
- **dim** (*int*) –Operation dimension. The value of *dim* must be in the range *[-input.ndim, input.ndim - 1]*.

Returns

tuple [Tensor], tuple of 2 Tensors, containing the cumulative minimum of elements and the index. The shape of each output tensor is the same as that of input *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *input* is a Tensor, but the type is complex or bool.
- **TypeError** –If *dim* is not an int.

- **ValueError** –If *dim* is out the range of *[-input.ndim, input.ndim - 1]*.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> import mindspore
>>> a = Tensor([-0.2284, -0.6628,  0.0975,  0.2680, -1.3298, -0.4220], mindspore.float32)
>>> output = mint.cummin(a, dim=0)
>>> print(output[0])
[-0.2284 -0.6628 -0.6628 -0.6628 -1.3298 -1.3298]
>>> print(output[1])
[0 1 1 1 4 4]
```

mindspore.mint.cumsum

`mindspore.mint.`**`cumsum`**(*input*, *dim*, *dtype=None*)

Computes the cumulative sum of input Tensor along *dim*.

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Parameters

- **input** (*Tensor*) –The input Tensor.
- **dim** (*int*) –Dim along which the cumulative sum is computed.
- **dtype** (*mindspore.dtype*, optional) –The desired dtype of returned Tensor. If specified, the input Tensor will be cast to *dtype* before the computation. This is useful for preventing overflows. If not specified, stay the same as original Tensor. Default: `None` .

Returns

Tensor, the shape of the output Tensor is consistent with the input Tensor's.

Raises

- **TypeError** –If *input* is not a Tensor.
- **ValueError** –If the *dim* is out of range.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> x = Tensor(np.array([[3, 4, 6, 10], [1, 6, 7, 9], [4, 3, 8, 7], [1, 3, 7, 9]]).astype(np.
↪float32))
>>> # case 1: along the dim 0
>>> y = mint.cumsum(x, 0)
>>> print(y)
[[ 3.  4.  6. 10.]
[ 4. 10. 13. 19.]
[ 8. 13. 21. 26.]
[ 9. 16. 28. 35.]]
>>> # case 2: along the dim 1
>>> y = mint.cumsum(x, 1)
>>> print(y)
[[ 3.  7. 13. 23.]
[ 1.  7. 14. 23.]
[ 4.  7. 15. 22.]
[ 1.  4. 11. 20.]]
```

mindspore.mint.diag

mindspore.mint.**diag**(*input*, *diagonal=0*)

If input is a vector (1-D tensor), then returns a 2-D square tensor with the elements of input as the diagonal.

If input is a matrix (2-D tensor), then returns a 1-D tensor with the diagonal elements of input.

The argument diagonal controls which diagonal to consider:

- If *diagonal* = 0, it is the main diagonal.
- If *diagonal* > 0, it is above the main diagonal.
- If *diagonal* < 0, it is below the main diagonal.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input tensor.
- **diagonal** (*int, optional*) –the diagonal to consider. Defaults: 0.

Returns

Tensor, has the same dtype as the *input*, its shape is up to *diagonal*.

- If *input* shape is (x_0) : then output shape is $(x_0 + |diagonal|, x_0 + |diagonal|)$ 2-D Tensor.
- If *input* shape is (x_0, x_1) : then output shape is main diagonal to move $(|diagonal|)$ elements remains elements' length 1-D Tensor.

Raises

- **TypeError** –If *input* is not a Tensor.

- **ValueError** –If shape of *input* is not 1-D and 2-D.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> input = Tensor([1, 2, 3, 4]).astype('int32')
>>> output = mint.diag(input)
>>> print(output)
[[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
```

mindspore.mint.flatten

`mindspore.mint.`**`flatten`**(*input*, *start_dim=0*, *end_dim=- 1*)

Flatten a tensor along dimensions from *start_dim* to *end_dim*.

Parameters

- **input** (`Tensor`) –The input Tensor.
- **start_dim** (*int, optional*) –The first dimension to flatten. Default: `0` .
- **end_dim** (*int, optional*) –The last dimension to flatten. Default: `-1` .

Returns

Tensor. If no dimensions are flattened, returns the original *input*, otherwise return the flattened Tensor. If *input* is a 0-dimensional Tensor, a 1-dimensional Tensor will be returned.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *start_dim* or *end_dim* is not int.
- **ValueError** –If *start_dim* is greater than *end_dim* after canonicalized.
- **ValueError** –If *start_dim* or *end_dim* is not in range of [-input.dim, input.dim-1].

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.ones(shape=[1, 2, 3, 4]), mindspore.float32)
>>> output = mint.flatten(input_x)
>>> print(output.shape)
(24,)
```

mindspore.mint.flip

mindspore.mint.**flip**(*input*, *dims*)

Reverses elements in a tensor along the given dims.

Parameters

- **input** (Tensor) –The input tensor.
- **dims** (*Union[list[int], tuple[int]]*) –The dimension to flip.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor(mindspore.ops.arange(1, 9).reshape((2, 2, 2)))
>>> output = mindspore.mint.flip(input, (0, 2))
>>> print(output)
[[[6 5]
  [8 7]]
 [[2 1]
  [4 3]]]
```

mindspore.mint.repeat_interleave

mindspore.mint.**repeat_interleave**(*input*, *repeats*, *dim=None*, ***, *output_size=None*) → *Tensor*

Repeat elements of a tensor along an axis, like *mindspore.numpy.repeat()*.

> **Warning:** Only support on Atlas A2 training series.

Parameters

- **input** (Tensor) –The tensor to repeat values for. Must be of types: float16, float32, int8, uint8, int16, int32, or int64.
- **repeats** (*Union[int, tuple, list, Tensor]*) –The number of times to repeat, must be positive.

- **dim** (*int, optional*) –The dim along which to repeat, Default: `None`. If dims is None, the input Tensor will be flattened and the output will alse be flattened.

Keyword Arguments

output_size (*int, optional*) –Total output size for the given axis (e.g. sum of repeats), Default: `None`.

Returns

One tensor with values repeated along the specified dim. If input has shape $(s1, s2, ..., sn)$ and dim is i, the output will have shape $(s1, s2, ..., si * repeats, ..., sn)$. The output type will be the same as the type of *input*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[0, 1, 2], [3, 4, 5]]), mindspore.int32)
>>> output = mint.repeat_interleave(input, repeats=2, dim=0)
>>> print(output)
[[0 1 2]
 [0 1 2]
 [3 4 5]
 [3 4 5]]
```

mindspore.mint.searchsorted

`mindspore.mint.`**searchsorted**(*sorted_sequence*, *values*, *, *out_int32=False*, *right=False*, *side=None*, *sorter=None*)

Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.

Parameters

- **sorted_sequence** (`Tensor`) –The input tensor. If *sorter* not provided, it must contain a increasing sequence on the innermost dimension.
- **values** (`Tensor`) –The value that need to be inserted.

Keyword Arguments

- **out_int32** (`bool, optional`) –Whether the output datatype will be mindspore.int32. if `False` , the output datatype will be mindspore.int64. Default `False` .
- **right** (`bool, optional`) –Search Strategy. If `True` , return the last suitable index found; if `False` , return the first such index. Default `False` .
- **side** (*str, optional*) –the same as right but preferred. `left` corresponds to `False` for *right* and `right` corresponds to `True` for *right*. An error will be reported if this parameter is set to `left` while *right* is `True`. Default `None` .
- **sorter** (`Tensor, optional`) –An index sequence sorted in ascending order along the innermost dimension of *sorted_sequence* , which is used together with the unsorted *sorted_sequence* . CPU and GPU only support `None`. Default `None` .

Returns

Tensor

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> sorted_sequence = mindspore.tensor([1, 2, 2, 3, 4, 5, 5])
>>> values = mindspore.tensor([2])
>>> mindspore.mint.searchsorted(sorted_sequence, values)
Tensor(shape=[1], dtype=Int64, value= [1])
```

mindspore.mint.triangular_solve

`mindspore.mint.`**`triangular_solve`**(*b*, *A*, *upper=True*, *transpose=False*, *unitriangular=False*)

Solves a system of equations with a square upper or lower triangular invertible matrix A and multiple right-hand sides b.

In symbols, it solves $AX = b$ and assumes A is square upper-triangular (or lower-triangular if `upper = False`) and does not have zeros on the diagonal.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **b** (*Tensor*) – A Tensor of shape $(*, M, K)$ where * is zero of more batch dimensions.
- **A** (*Tensor*) – A Tensor of shape $(*, M, M)$ where * is zero of more batch dimensions.
- **upper** (*bool, optional*) – Whether A is upper or lower triangular. Default: `True`.
- **transpose** (*bool, optional*) – Solves $op(A)X = b$ where $op(A) = A^T$ if this flag is True, and $op(A) = A$ if it is False, Default: `False`.
- **unitriangular** (*bool, optional*) – Whether A is unit triangular. If True, the diagonal elements of A are assumed to be 1 and not referenced from A. Default: `False`.

Returns

A tuple of X and A.

Raises

- **TypeError** – If argument b is not Tensor.
- **TypeError** – If argument A is not Tensor.
- **TypeError** – If *upper* is not bool.
- **TypeError** – If *transpose* is not bool.
- **TypeError** – If *unitriangular* is not bool.
- **ValueError** – If the rank of b or A is not in the range of $[2, 6]$.
- **ValueError** – If the shapes of b and A are not matched.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import mint
>>> from mindspore import Tensor
>>> b = Tensor(np.ones((2, 3, 4), dtype=np.float32))
>>> A = Tensor(np.ones((2, 3, 3), dtype=np.float32))
>>> output = mint.triangular_solve(b, A)
>>> print(output[0])
[[[ 0.  0.  0.  0.]
  [ 0.  0.  0.  0.]
  [ 1.  1.  1.  1.]]
 [[ 0.  0.  0.  0.]
  [ 0.  0.  0.  0.]
  [ 1.  1.  1.  1.]]]
```

4.4 mindspore.mint.nn

4.4.1 Convolution Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.Conv2d	2D convolution layer.	Ascend
mindspore.mint.nn.Conv3d	3D convolution layer.	Ascend
mindspore.mint.nn.ConvTranspose2d	Applies a 2D transposed convolution operator over an input image composed of several input planes.	Ascend
mindspore.mint.nn.Fold	Combines an array of sliding local blocks into a large containing tensor.	Ascend
mindspore.mint.nn.Unfold	Extracts sliding local blocks from a batched input tensor.	Ascend

mindspore.mint.nn.Conv2d

class mindspore.mint.nn.**Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *dtype=None*)

2D convolution layer.

Applies a 2D convolution over an input tensor which is typically of shape $(N, C_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, H is feature height, W is feature width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where *bias* is the output channel bias, *ccor* is the cross-correlation, *weight* is the convolution kernel value and X represents the input feature map.

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.

- j corresponds to the output channel, the range is $[0, C_{out} - 1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in} - 1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1]), where kernel_size[0] and kernel_size[1] are the height and width of the kernel, respectively. If we consider the input and output channels as well as the *groups* parameter, the complete kernel shape will be (C_{out}, C_{in}/groups, kernel_size[0], kernel_size[1]), where *groups* is the number of groups dividing *x*'s input channel when applying groups convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Parameters

- **in_channels** (*int*) −The channel number of the input tensor of the Conv2d layer.
- **out_channels** (*int*) −The channel number of the output tensor of the Conv2d layer.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) −Specifies the height and width of the 2D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the height and width of the convolution kernel. A tuple of two integers represents the height and width of the convolution kernel respectively.
- **stride** (*Union[int, tuple[int], list[int]], optional*) −The movement stride of the 2D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the movement step size in both height and width directions. A tuple of two integers represents the movement step size in the height and width directions respectively. Default: `1` .
- **padding** (*Union[int, tuple[int], list[int], str], optional*) −The number of padding on the height and width directions of the input. The data type is an integer or a tuple of two integers or string {`"valid"`, `"same"`}. If *padding* is an integer, then *padding_{H}* and *padding_{W}* are all equal to *padding*. If *padding* is a tuple of 2 integers, then *padding_{H}* and *padding_{W}* is equal to *padding[0]* and *padding[1]* respectively. The value should be greater than or equal to 0. Default: `0` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *stride* must be 1.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded.
- **padding_mode** (*str, optional*) −Specifies the padding mode with a padding value of 0. It can be set to: `"zeros"` , `"reflect"` `"circular"` or `"replicate"` . Default: `"zeros"` .
- **dilation** (*Union[int, tuple[int], list[int]], optional*) −Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 2 or 4 integers. A single int means the dilation size is the same in both the height and width directions. A tuple of two ints represents the dilation size in the height and width directions, respectively. For a tuple of four ints, the two ints correspond to (N, C) dimension are treated as 1, and the two correspond to (H, W) dimensions is the dilation size in the height and width directions respectively. Assuming $dilation = (d0, d1)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the height direction and $d1 - 1$ elements in the width direction. The values in the height and width dimensions are in the ranges [1, H] and [1, W], respectively. Default: `1` .
- **groups** (*int, optional*) −Splits filter into groups, *in_channels* and *out_channels* must be divisible by *groups*. If the groups is equal to *in_channels* and *out_channels*, this 2D convolution layer also can be called 2D depthwise convolution layer. Default: `1` . The following restraints must be met:
  - (C_{in}

– (C_{out}
– (C_{out} >= groups)
– (kernel_size[1] = C_{in}/groups)

- **bias** (*bool, optional*) –Whether the Conv2d layer has a bias parameter. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `None`, using `mstype.float32`.

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}).

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$.

padding is `'same'`:

$$\begin{aligned} H_{out} &= \left\lceil \frac{H_{in}}{\text{stride}[0]} \right\rceil \\ W_{out} &= \left\lceil \frac{W_{in}}{\text{stride}[1]} \right\rceil \end{aligned}$$

padding is `'valid'`:

$$\begin{aligned} H_{out} &= \left\lfloor \frac{H_{in} - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \end{aligned}$$

padding is int or tuple/list:

$$\begin{aligned} H_{out} &= \left\lfloor \frac{H_{in} + padding[0] + padding[1] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} \right\rfloor + 1 \\ W_{out} &= \left\lfloor \frac{W_{in} + padding[2] + padding[3] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} \right\rfloor + 1 \end{aligned}$$

Raises

- **ValueError** –Args and size of the input feature map should satisfy the output formula to ensure that the size of the output feature map is positive; otherwise, an error will be reported.
- **RuntimeError** –On Ascend, due to the limitation of the L1 cache size of different NPU chip, if input size or kernel size is too large, it may trigger an error.
- **TypeError** –If *in_channels*, *out_channels* or *groups* is not an int.
- **TypeError** –If *kernel_size*, *stride* or *dilation* is neither an int nor a tuple.
- **ValueError** –If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** –If *padding* is less than 0.
- **ValueError** –If *padding* is *same* , *stride* is not equal to 1.
- **ValueError** –The input parameters do not satisfy the convolution output formula.
- **ValueError** –The *kernel_size* cannot exceed the size of the input feature map.
- **ValueError** –The value of padding cannot cause the calculation area to exceed the input size.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> net = mint.nn.Conv2d(120, 240, 4, bias=False)
>>> x = Tensor(np.ones([1, 120, 1024, 640]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 240, 1021, 637)
```

mindspore.mint.nn.Conv3d

class mindspore.mint.nn.**Conv3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *dtype=None*)

3D convolution layer.

Applies a 3D convolution over an input tensor. The input tensor is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, D, H, W are the depth, height and width of the feature graph, respectively.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1], kernel_size[2]), where kernel_size[0], kernel_size[1] and kernel_size[2] are the depth, height and width of the kernel, respectively. If we consider the input and output channels as well as the *groups* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{groups}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$, where *groups* is the number of groups dividing x's input channel when applying groups convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

For the detail of limitations of the parameters, please refer to `mindspore.mint.nn.functional.conv3d()`.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **in_channels** (*int*) –The channel number of the input tensor of the Conv3d layer.

- **out_channels** (*int*) −The channel number of the output tensor of the Conv3d layer.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) −Specifies the height and width of the 3D convolution kernel. The data type is an integer or a tuple of two integers. An integer represents the height and width of the convolution kernel. A tuple of two integers represents the height and width of the convolution kernel respectively.
- **stride** (*Union[int, tuple[int], list[int]], optional*) −The movement stride of the 3D convolution kernel. The data type is an integer or a tuple of three integers. An integer represents the movement step size in both height and width directions. A tuple of three integers represents the movement step size in the depth, height and width directions respectively. Default: `1` .
- **padding** (*Union[int, tuple[int], list[int], str], optional*) −The number of padding on the depth, height and width directions of the input. The data type is an integer or string {`"valid"`, `"same"`} or a tuple of three integers. The value should be greater than or equal to 0. Default: `0` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *padding* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *padding* must be 0.
- **padding_mode** (*str, optional*) −Specifies the padding mode with a padding value of 0. It can be set to: `"zeros"` , `"reflect"` `"circular"` or `"replicate"` . Default: `"zeros"` .
- **dilation** (*Union[int, tuple[int], list[int]], optional*) −Controlling the space between the kernel points. Default: `1` .
- **groups** (*int, optional*) −Splits filter into groups, *in_channels* and *out_channels* must be divisible by *groups*. If the groups is equal to *in_channels* and *out_channels*. Default: `1` .
- **bias** (*bool, optional*) −Whether the Conv3d layer has a bias parameter. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) −Dtype of Parameters. Default: `None`, using `mstype.float32`.

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$ or $(C_{in}, D_{in}, H_{in}, W_{in})$.

Outputs:

Tensor of shape $(N, C_{out}, D_{out}, H_{out}, W_{out})$ or $(C_{out}, D_{out}, H_{out}, W_{out})$.

padding is `"same"`:

$$
\begin{aligned}
D_{out} &= \left\lceil \frac{D_{in}}{\text{stride}[0]} \right\rceil \\
H_{out} &= \left\lceil \frac{H_{in}}{\text{stride}[1]} \right\rceil \\
W_{out} &= \left\lceil \frac{W_{in}}{\text{stride}[2]} \right\rceil
\end{aligned}
$$

padding is `"valid"`:

$$
\begin{aligned}
D_{out} &= \left\lceil \frac{D_{in} - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1)}{\text{stride}[0]} \right\rceil \\
H_{out} &= \left\lceil \frac{H_{in} - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1)}{\text{stride}[1]} \right\rceil \\
W_{out} &= \left\lceil \frac{W_{in} - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1)}{\text{stride}[2]} \right\rceil
\end{aligned}
$$

padding is int or tuple/list:

$$
\begin{aligned}
D_{out} &= \left\lfloor \frac{D_{in}+padding[0]+padding[1]-\text{dilation}[0]\times(\text{kernel\_size}[0]-1)-1}{\text{stride}[0]} + 1 \right\rfloor \\
H_{out} &= \left\lfloor \frac{H_{in}+padding[2]+padding[3]-\text{dilation}[1]\times(\text{kernel\_size}[1]-1)-1}{\text{stride}[1]} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W_{in}+padding[4]+padding[5]-\text{dilation}[2]\times(\text{kernel\_size}[2]-1)-1}{\text{stride}[2]} + 1 \right\rfloor
\end{aligned}
$$

Raises

- **TypeError** – If *in_channels*, *out_channels* or *groups* is not an int.
- **TypeError** – If *kernel_size*, *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **ValueError** – If *in_channels*, *out_channels*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *padding* is less than 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> net = mint.nn.Conv3d(120, 10, 4)
>>> x = Tensor(np.ones([1, 120, 10, 23, 34]), mindspore.float32)
>>> output = net(x).shape
>>> print(output)
(1, 10, 7, 20, 31)
```

mindspore.mint.nn.ConvTranspose2d

class mindspore.mint.nn.**ConvTranspose2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*, *dtype=None*)

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation as it does not compute a true inverse of convolution).

The parameters *kernel_size*, *stride*, *padding*, *output_padding* can either be:

- a single `int` – in which case the same value is used for the height and width dimensions
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

Warning:

- This is an experimental API that is subject to change or deletion.
- In the scenario where inputs are non-contiguous, *output_padding* must be less than *stride* .
- For Atlas training products, when the dtype of input is float32, the *groups* only supports 1.

Parameters

- **in_channels** (*int*) –Number of channels in the input image.
- **out_channels** (*int*) –Number of channels produced by the convolution.
- **kernel_size** (*Union[int, tuple(int)]*) –Size of the convolving kernel.
- **stride** (*Union[int, tuple(int)], optional*) –Stride of the convolution. Default: `1` .
- **padding** (*Union[int, tuple(int)], optional*) –$dilation * (kernel\_size - 1) - padding$ zero-padding will be added to both sides of each dimension in the input. Default: `0` .
- **output_padding** (*Union[int, tuple(int)], optional*) –Additional size added to one side of each dimension in the output shape. The value of *output_padding* must be less than *stride* or *dilation* . Default: `0` .
- **groups** (*int, optional*) –Number of blocked connections from input channels to output channels. Default: `1`
- **bias** (*bool, optional*) –If `True`, adds a learnable bias to the output. Default: `True` .
- **dilation** (*Union[int, tuple(int)], optional*) –Spacing between kernel elements. Default: `1` .
- **padding_mode** (*str, optional*) –Specifies the padding mode with a padding value. For now, it can only be set to: `"zeros"`. Default: `"zeros"` .
- **dtype** (*mindspore.dtype, optional*) –Dtype of Parameters. Default: `None` , when it's `None` , the dtype of Parameters would be mstype.float32.

Variables:

- **weigh** (Parameter) - the learnable weights of the module of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kernel_size[0], kernel_size[1]) . The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (Parameter) - the learnable bias of the module of shape (out_channels,) . If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$.

Inputs:

- **input** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}) .

Outputs:

Tensor of shape $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> # With square kernels and equal stride
>>> m = mint.nn.ConvTranspose2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = mint.nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> input = mint.randn(20, 16, 50, 100)
```

(continues on next page)

(continued from previous page)

```
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = mint.randn(1, 16, 12, 12)
>>> downsample = mint.nn.Conv2d(16, 16, 3, stride=2, padding=1)
>>> upsample = mint.nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
>>> h = downsample(input)
>>> h.shape
(1, 16, 6, 6)
>>> output = upsample(h, output_size=input.shape)
>>> output.shape
(1, 16, 12, 12)
```

mindspore.mint.nn.Fold

class mindspore.mint.nn.**Fold**(*output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Combines an array of sliding local blocks into a large containing tensor.

For details, please refer to *mindspore.mint.nn.functional.fold()*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> fold = mint.nn.Fold([8, 8], [2, 2], [2, 2], [2, 2], [2, 2])
>>> input = Tensor(input_data=np.random.rand(16, 64, 25), dtype=mstype.float32)
>>> output = fold(input)
>>> print(output.shape)
(16, 16, 8, 8)
```

mindspore.mint.nn.Unfold

class mindspore.mint.nn.**Unfold**(*kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Extracts sliding local blocks from a batched input tensor.

For details, please refer to *mindspore.mint.nn.functional.unfold()*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.random.rand(4, 4, 32, 32), mindspore.float64)
>>> unfold = mint.nn.Unfold(kernel_size=3, dilation=1, stride=1)
>>> output = unfold(input)
>>> print(output.shape)
(4, 36, 900)
```

4.4.2 Normalization Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.BatchNorm1d	Applies Batch Normalization over a 2D or 3D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .	Ascend
mindspore.mint.nn.BatchNorm2d	Applies Batch Normalization over a 4D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .	Ascend
mindspore.mint.nn.BatchNorm3d	Applies Batch Normalization over a 5D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .	Ascend
mindspore.mint.nn.GroupNorm	Group Normalization over a mini-batch of inputs.	Ascend
mindspore.mint.nn.LayerNorm	Applies Layer Normalization over a mini-batch of inputs.	Ascend
mindspore.mint.nn.SyncBatchNorm	Sync Batch Normalization layer over a N-dimension input.	Ascend

mindspore.mint.nn.BatchNorm1d

class mindspore.mint.nn.**BatchNorm1d**(*num_features: int*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *dtype=None*)

Applies Batch Normalization over a 2D or 3D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the number of features or channels of the input). By default, the elements of γ are set to 1 and the elements of β are set to 0.

Warning: This API does not support Dynamic Rank. This is an experimental API that is subject to change or deletion.

Parameters

- **num_features** (*int*) –C from an expected input of shape (N, C, L).
- **eps** (*float, optional*) –a value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float, optional*) –the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average. Default: `0.1` .
- **affine** (*bool, optional*) –a boolean value that when set to `True`, this cell has learnable affine parameters. Default: `True` .
- **track_running_stats** (*bool, optional*) –a boolean value that when set to `True`, this cell tracks the running mean and variance, and when set to `False`, this cell does not track such statistics. And this cell always uses batch statistics in both training and eval modes. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `None` .

Inputs:

- **input** (Tensor) - The input with shape (N, C) or (N, C, L), where N means batch, C means the number of feature or the number of channel, and L is the length of sequence.

Outputs:

Tensor, has the same type and shape as *input*.

Raises

- **TypeError** –If *num_features* is not an int number.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_features* is less than 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input_x = mindspore.Tensor([[0.7, 0.5, 0.5, 0.6], [0.5, 0.4, 0.6, 0.9]])
>>> net = mint.nn.BatchNorm1d(4)
>>> output = net(input_x)
>>> print(output)
[[ 0.99950075 0.9980011 -0.9980068 -0.9997783]
 [-0.9995012 -0.99799967 0.9980068  0.9997778]]
```

mindspore.mint.nn.BatchNorm2d

class mindspore.mint.nn.**BatchNorm2d**(*num_features: int*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *dtype=None*)

Applies Batch Normalization over a 4D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the number of features or channels of the input). By default, the elements of γ are set to 1 and the elements of β are set to 0.

Warning:

- This API does not support Dynamic Rank.
- This is an experimental API that is subject to change or deletion.

Parameters

- **num_features** (*int*) – C from an expected input of shape (N, C, H, W).
- **eps** (*float, optional*) – a value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float, optional*) – the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average. Default: `0.1` .
- **affine** (*bool, optional*) – a boolean value that when set to `True`, this cell has learnable affine parameters. Default: `True` .
- **track_running_stats** (*bool, optional*) – a boolean value that when set to `True`, this cell tracks the running mean and variance, and when set to `False`, this cell does not track such statistics. And this cell always uses batch statistics in both train and eval modes. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) – Dtype of Parameters. Default: `None` .

Inputs:

- **input** (Tensor) - The input with shape (N, C, H, W).

Outputs:

Tensor, has the same type and shape as *input*.

Raises

- **TypeError** – If *num_features* is not an int number.
- **TypeError** – If *eps* is not a float.
- **ValueError** – If *num_features* is less than 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input_x = mindspore.Tensor([0.3, 0.4, 0.5, 0.3])
>>> input_x = input_x.reshape((2, 2, 1, 1))
>>> net = mint.nn.BatchNorm2d(2)
>>> output = net(input_x)
>>> print(output)
[[[[-0.99950075]]
  [[0.9980087]]]
  [[[0.999501]]
  [[-0.9980097]]]]
```

mindspore.mint.nn.BatchNorm3d

class mindspore.mint.nn.**BatchNorm3d**(*num_features: int*, *eps=1e-5*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *dtype=None*)

Applies Batch Normalization over a 5D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the number of features or channels of the input). By default, the elements of γ are set to 1 and the elements of β are set to 0.

> **Warning:** This API does not support Dynamic Rank. This is an experimental API that is subject to change or deletion.

Parameters

- **num_features** (*int*) –C from an expected input of shape (N, C, D, H, W).
- **eps** (*float, optional*) –a value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float, optional*) –the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average. Default: `0.1` .
- **affine** (*bool, optional*) –a boolean value that when set to `True`, this cell has learnable affine parameters. Default: `True` .
- **track_running_stats** (*bool, optional*) –a boolean value that when set to `True`, this cell tracks the running mean and variance, and when set to `False`, this cell does not track such statistics. And this cell always uses batch statistics in both training and eval modes. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `None` .

Inputs:

- **input** (Tensor) - The input with shape (N, C, D, H, W).

Outputs:

Tensor, has the same type and shape as *input*.

Raises

- **TypeError** –If *num_features* is not an int number.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_features* is less than 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input_x = mindspore.Tensor([0.1, 0.9, 1.2, 2.3])
>>> input_x = input_x.reshape((1, 2, 1, 1, 2))
>>> net = mint.nn.BatchNorm3d(2)
>>> output = net(input_x)
>>> print(output)
[[[[[-0.9999688 0.99996865]]]
  [[[-0.9999833 06.9999831]]]]]
```

mindspore.mint.nn.GroupNorm

class mindspore.mint.nn.**GroupNorm**(*num_groups*, *num_channels*, *eps=1e-05*, *affine=True*, *dtype=None*)

Group Normalization over a mini-batch of inputs.

Group Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Group Normalization.

Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization, and it performs very stable over a wide range of batch size. γ and β are trainable scale and shift. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

where γ is *weight*, β is *bias*, and ϵ is *eps*.

Parameters

- **num_groups** (*int*) –The number of groups to be divided along the channel dimension.
- **num_channels** (*int*) –The number of input channels.
- **eps** (*float, optional*) –A value added to the denominator for numerical stability. Default: `1e-05` .
- **affine** (*bool, optional*) –The parameters, such as γ and β, are learnable when set to `true` . Default: `True` .
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `None` .

Inputs:

- **input** (Tensor) - The input feature with shape $(N, C, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the *input*.

Raises

- **TypeError** – If *num_groups* or *num_channels* is not an int.
- **TypeError** – If *eps* is not a float.
- **TypeError** – If *affine* is not a bool.
- **ValueError** – If *num_groups* or *num_channels* is less than 1.
- **ValueError** – If *num_channels* is not divided by *num_groups*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> group_norm_op = ms.mint.nn.GroupNorm(2, 2)
>>> x = ms.Tensor(np.ones([1, 2, 4, 4], np.float32))
>>> output = group_norm_op(x)
>>> print(output)
[[[[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
  [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]]]
```

mindspore.mint.nn.LayerNorm

class mindspore.mint.nn.**LayerNorm**(*normalized_shape*, *eps=1e-5*, *elementwise_affine=True*, *bias=True*, *dtype=None*)

Applies Layer Normalization over a mini-batch of inputs.

Layer Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Layer Normalization.

Unlike Batch Normalization, Layer Normalization performs exactly the same computation at training and testing time. It is applied across all channels and pixel but only one batch size. γ is the scale value learned through training and β is the shift value. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **normalized_shape** (*Union(tuple[int], list[int], int)*) – The normalized shape of x for LayerNorm.
- **eps** (*float, optional*) – A value added to the denominator for numerical stability(ϵ). Default: `1e-5` .

- **elementwise_affine** (*bool, optional*) –Whether affine transformation is required. When this parameter is set to `True`, the weight parameter is initialized to 1 and the offset is initialized to 0. Default: `True`.
- **bias** (*bool, optional*) –If set to `False`, the layer will not learn an additive bias (only relevant if *elementwise_affine* is `True`). Default: `True`.
- **dtype** (*mindspore.dtype*, optional) –Dtype of Parameters. Default: `None` .

Inputs:

- **x** (Tensor) - The shape is $(N, *)$, where $*$ is equal to normalized_shape.

Outputs:

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the *x*.

Raises

TypeError –If *eps* is not a float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> x = ms.Tensor(np.ones([20, 5, 10, 10]), ms.float32)
>>> shape1 = x.shape[1:]
>>> m = ms.mint.nn.LayerNorm(shape1)
>>> output = m(x).shape
>>> print(output)
(20, 5, 10, 10)
```

mindspore.mint.nn.SyncBatchNorm

class mindspore.mint.nn.**SyncBatchNorm**(*num_features: int*, *eps: float = 1e-5*, *momentum: float = 0.1*, *affine: bool = True*, *track_running_stats: bool = True*, *process_group: Optional[str] = None*, *dtype=None*)

Sync Batch Normalization layer over a N-dimension input.

Sync Batch Normalization is cross device synchronized Batch Normalization. The implementation of Batch Normalization only normalizes the data within each device. Sync Batch Normalization will normalize the input within the group. It has been described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the feature using a mini-batch of data and the learned parameters which can be described in the following formula.

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **num_features** (*int*) –*C* from an expected input of size $(N, C, +)$.

- **eps** (*float, optional*) – ϵ, a value added to the denominator for numerical stability. Default: `1e-5` .
- **momentum** (*float, optional*) – A floating hyperparameter of the momentum for the running_mean and running_var computation. Default: `0.1` .
- **affine** (*bool, optional*) – A bool value. When set to `True` , γ and β are learnable parameters. When set to `False` , γ and β are unlearnable parameters. Default: `True` .
- **track_running_stats** (*bool, optional*) – a boolean value that when set to `True`, this cell tracks the running mean and variance, and when set to `False`, this cell does not track such statistics. And this cell always uses batch statistics in both training and eval modes. Default: `True` .
- **process_group** (*str, optional*) – synchronization of stats happen within each process group individually. Default behavior is synchronization across the whole world. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) – Dtype of Parameters. Default: `None` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, +)$.

Outputs:

Tensor, the normalized, scaled, offset tensor, of shape $(N, C_{out}, +)$.

Raises

- **TypeError** – If *num_features* is not an int.
- **TypeError** – If *eps* is not a float.
- **ValueError** – If *num_features* is less than 1.
- **ValueError** – If *momentum* is not in range [0, 1].
- **ValueError** – If rank_id in *process_group* is not in range [0, rank_size).

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Here, examples use msrun to pull multi-process distributed tasks across nodes with a single command line instruction. Please see the Ascend tutorial for more details.

This example should be run with multiple devices.

```
>>> # Firstly, preparing test_syncbn.py:
>>> import numpy as np
>>> import mindspore
>>> import mindspore.context as context
>>> from mindspore.mint.nn.layer import SyncBatchNorm
>>> from mindspore import Tensor
>>> from mindspore.communication import init, create_group, get_local_rank
>>> init()
>>> context.set_context(mode=context.PYNATIVE_MODE, device_target="Ascend")
```

(continues on next page)

(continued from previous page)

```
>>> group = "0-1"
>>> rank_ids = [0, 1]
>>> create_group(group, rank_ids)
>>> sync_batch_norm = SyncBatchNorm(num_features=2, process_group=group, dtype=mindspore.
↪float32)
>>> sync_batch_norm.set_train(False)
>>> input_x = Tensor(np.linspace(0, 5, 2*2*2*2), mindspore.float32).reshape(2, 2, 2, 2)
>>> output_data = sync_batch_norm(input_x)
>>> # Then, executing the command such as the following:
>>> # msrun --worker_num=2 --local_worker_num=2 --master_port=8975 --log_dir=msrun_log --
↪join=True
>>> # --cluster_time_out=100 pytest -s -v test_syncbn.py
```

4.4.3 Non-linear Activations (weighted sum, nonlinearity)

API Name	Description	Supported Platforms
mindspore.mint.nn.ELU	Exponential Linear Unit activation function	Ascend
mindspore.mint.nn.GELU	Activation function GELU (Gaussian Error Linear Unit).	Ascend
mindspore.mint.nn.GLU	Computes GLU (Gated Linear Unit activation function) of the input tensor.	Ascend
mindspore.mint.nn.Hardshrink	Applies Hard Shrink activation function element-wise.	Ascend
mindspore.mint.nn.Hardsigmoid	Applies Hard Sigmoid activation function element-wise.	Ascend
mindspore.mint.nn.Hardswish	Applies Hard Swish activation function element-wise.	Ascend
mindspore.mint.nn.LogSigmoid	Applies logsigmoid activation element-wise.	Ascend
mindspore.mint.nn.LogSoftmax	Applies the Log Softmax function to the input tensor on the specified axis.	Ascend
mindspore.mint.nn.Mish	Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.	Ascend
mindspore.mint.nn.PReLU	Applies PReLU activation function element-wise.	Ascend
mindspore.mint.nn.ReLU	Applies ReLU (Rectified Linear Unit activation function) element-wise.	Ascend
mindspore.mint.nn.ReLU6	Activation function ReLU6.	Ascend
mindspore.mint.nn.SELU	Activation function SELU (Scaled exponential Linear Unit).	Ascend
mindspore.mint.nn.SiLU	Calculates the SiLU activation function element-wise.	Ascend
mindspore.mint.nn.Sigmoid	Applies sigmoid activation function element-wise.	Ascend
mindspore.mint.nn.Softmax	Applies the Softmax function to an n-dimensional input Tensor.	Ascend
mindspore.mint.nn.Softshrink	Applies the Softshrink function element-wise.	Ascend
mindspore.mint.nn.Tanh	Applies the Tanh function element-wise, returns a new tensor with the hyperbolic tangent of the elements of input.	Ascend

mindspore.mint.nn.ELU

class mindspore.mint.nn.**ELU**(*alpha=1.0*, *inplace=False*)

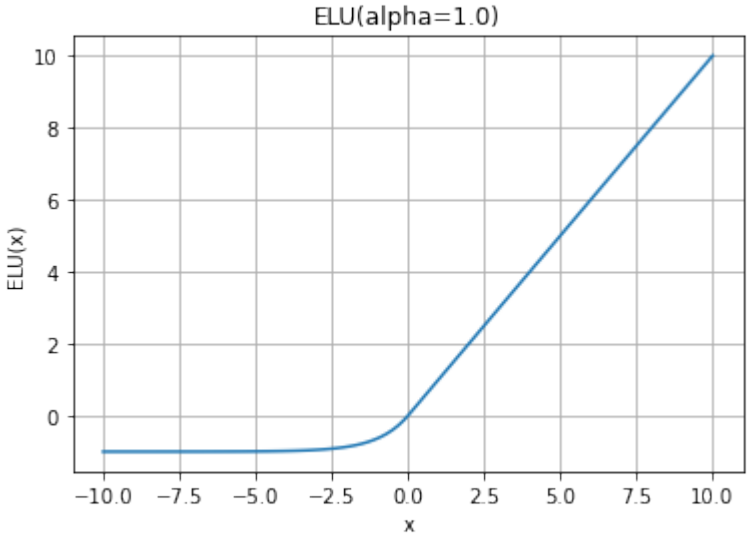
Exponential Linear Unit activation function

Applies the exponential linear unit function element-wise.The activation function is defined as:

$$ELU_i = \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \alpha * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where x_i represents the element of the input and α represents the *alpha* parameter, and *alpha* represents the smoothness of the ELU.

ELU Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **alpha** (*float, optional*) – The alpha value of ELU, the data type is float. Default: `1.0`.
- **inplace** (*bool, optional*) – Whether to use inplace mode, the data type is bool. Default: `False`.

Inputs:

- **input** (Tensor) - The input of ELU is a Tensor of any dimension.

Outputs:

Tensor, with the same shape and type as the *input*.

Raises

- **RuntimeError** – If the dtype of *input* is not float16, float32 or bfloat16.
- **TypeError** – If the dtype of *alpha* is not float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float32)
>>> elu = mint.nn.ELU()
>>> result = elu(input)
>>> print(result)
[-0.63212055  -0.86466473  0.  2.  1.]
```

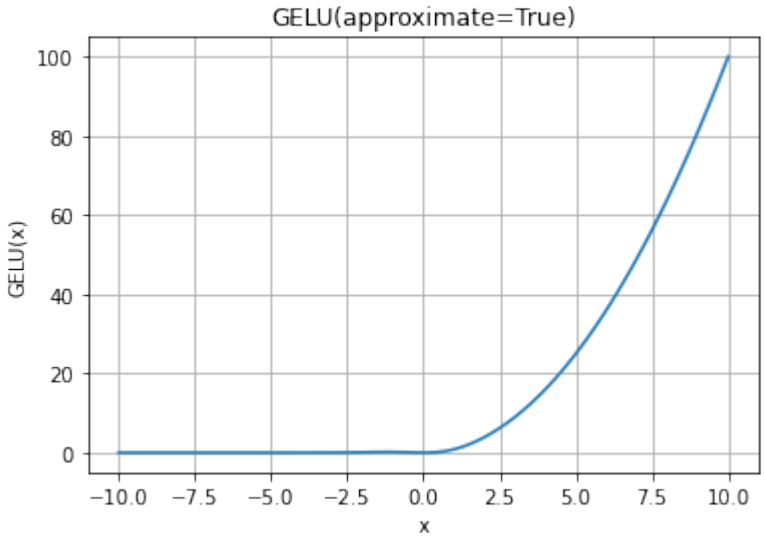
mindspore.mint.nn.GELU

class mindspore.mint.nn.**GELU**(*approximate='none'*)

Activation function GELU (Gaussian Error Linear Unit).

Refer to *mindspore.mint.nn.functional.gelu()* for more details.

GELU Activation Function Graph:



Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> gelu = mint.nn.GELU()
>>> output = gelu(input)
>>> print(output)
[[-1.58655241e-01  3.99987316e+00 -0.00000000e+00]
 [ 1.95449972e+00 -1.41860323e-06  9.0000000e+00]]
>>> gelu = mint.nn.GELU(approximate="tanh")
```

(continues on next page)

(continued from previous page)

```
>>> output = gelu(input)
>>> print(output)
[[-1.58808023e-01  3.99992990e+00 -3.10779147e-21]
 [ 1.95459759e+00 -2.29180174e-07  9.0000000e+00]]
```

mindspore.mint.nn.GLU

class mindspore.mint.nn.**GLU**(*dim=-1*)

Computes GLU (Gated Linear Unit activation function) of the input tensor.

$$GLU(a, b) = a \otimes \sigma(b)$$

where a is the first half of the *input* Tensor after *input* is split and b is the second half.

Here σ is the sigmoid function, and $\otimes$ is the Hadamard product. See Language Modeling with Gated Convluational Networks .

Parameters

dim (*int, optional*) – The dimension to split the input *input*. The value range is *[-r, r)* where *r* is the number of dimensions of *input*. Default: `-1` , the last dimension in *input*.

Inputs:

- **input** (Tensor) - Tensor to be calculated. Dtype is floating point and the shape is $(*_1, N, *_2)$ where * means, any number of additional dimensions. N is required to be an even number, where N is the size of *input* on the dimension selected by *dim*.

Outputs:

Tensor, the same dtype as the *input*, with the shape $(*_1, M, *_2)$ where $M = N/2$.

Raises

- **TypeError** – If *input* is not a Tensor or *dim* is not an int.
- **IndexError** – If the value of *dim* is out of the range of *[-r, r)*, where *r* is the number of dimensions of *input*.
- **RuntimeError** – If dtype of *input* is not supported.
- **RuntimeError** – If the length of *input* in the dimension selected by *dim* is not even.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import mint, Tensor
>>> glu = mint.nn.GLU()
>>> input = Tensor([[0.1, 0.2, 0.3, 0.4], [0.5, 0.6, 0.7, 0.8]])
>>> output = glu(input)
>>> print(output)
[[0.05744425 0.11973753]
 [0.33409387 0.41398472]]
```

mindspore.mint.nn.Hardshrink

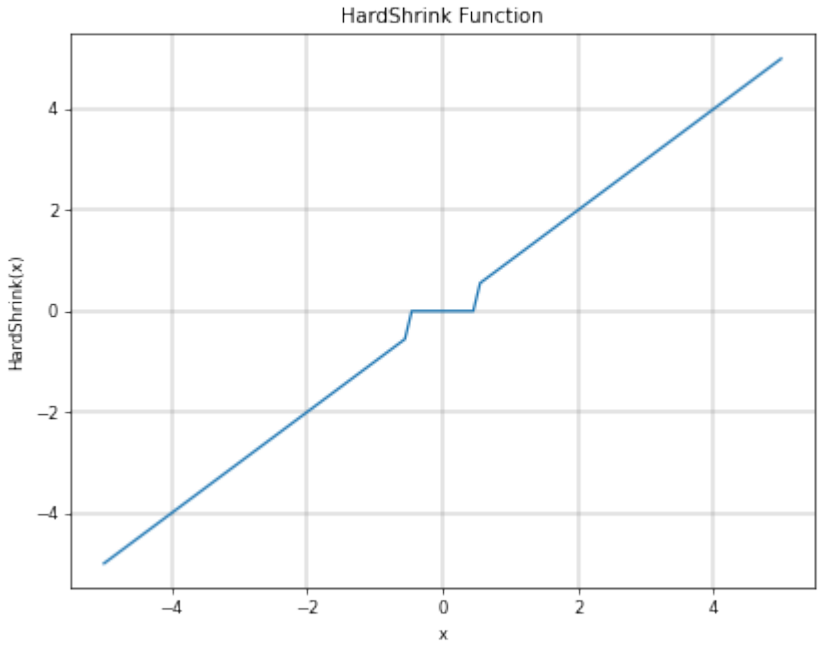
class mindspore.mint.nn.**Hardshrink**(*lambd=0.5*)

Applies Hard Shrink activation function element-wise.

The formula is defined as follows:

$$\text{Hardshrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

Hardshrink Activation Function Graph:



Parameters

lambd (*number, optional*) – The threshold λ defined by the Hard Shrink formula. Default: `0.5`.

Inputs:

- **input** (Tensor) - The input of Hard Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.

Outputs:

Tensor, the same shape and data type as the input.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[0.5, 1, 2.0], [0.0533, 0.0776, -2.1233]]), mindspore.float32)
>>> hshrink = mint.nn.Hardshrink()
>>> output = hshrink(input)
>>> print(output)
[[ 0.      1.      2.    ]
 [ 0.      0.     -2.1233]]
```

mindspore.mint.nn.Hardsigmoid

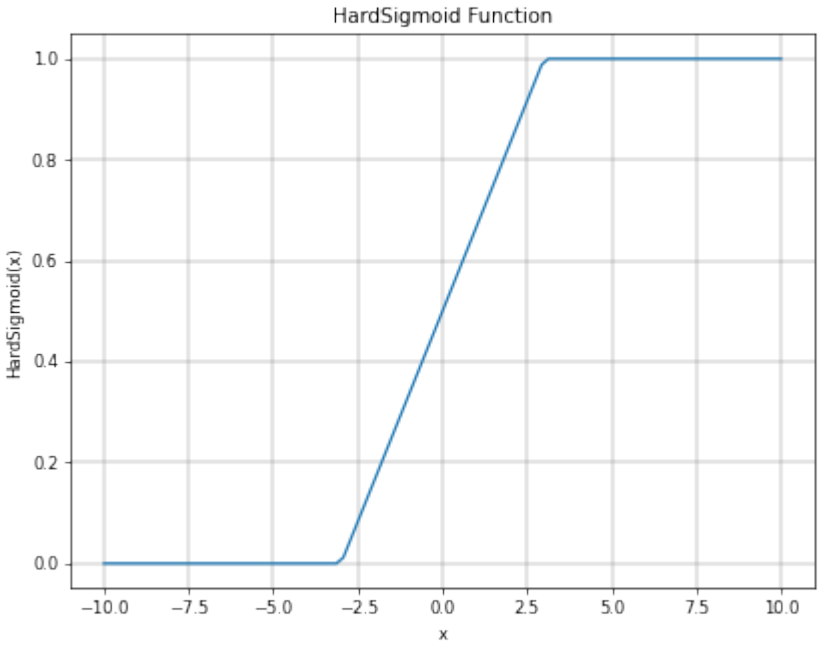
class mindspore.mint.nn.**Hardsigmoid**

Applies Hard Sigmoid activation function element-wise.

Hard Sigmoid is defined as:

$$\text{Hardsigmoid}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ 1, & \text{if } input \geq +3, \\ input/6 + 1/2, & \text{otherwise} \end{cases}$$

Hardsigmoid Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of Hardsigmoid.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> hsigmoid = mint.nn.Hardsigmoid()
>>> result = hsigmoid(input)
>>> print(result)
[0.3333 0.1666 0.5    0.8335 0.6665]
```

mindspore.mint.nn.Hardswish

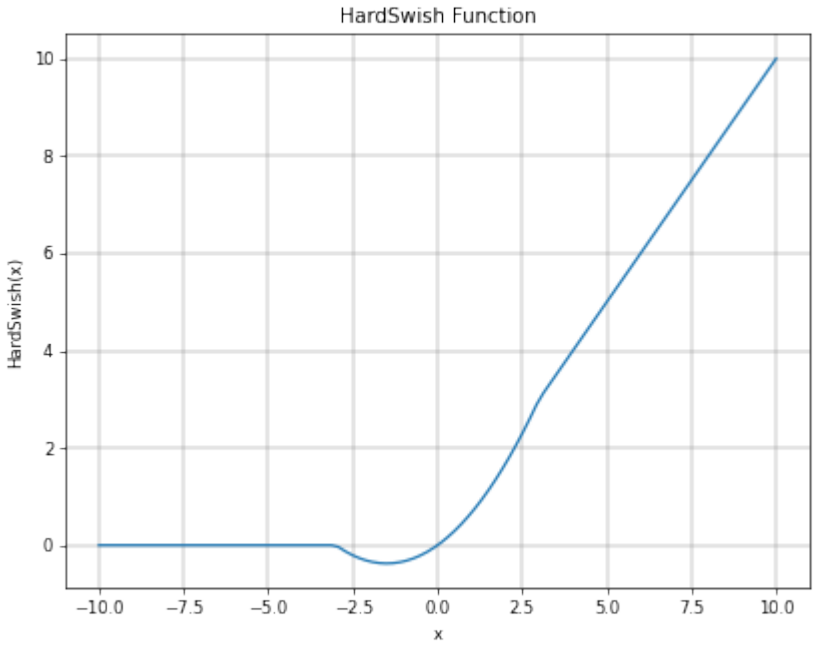
class mindspore.mint.nn.**Hardswish**

Applies Hard Swish activation function element-wise.

Hard swish is defined as:

$$\text{Hardswish}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ input, & \text{if } input \geq +3, \\ input * (input + 3)/6, & \text{otherwise} \end{cases}$$

Hardswish Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of Hardswish.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> hswish = mint.nn.Hardswish()
>>> result = hswish(input)
>>> print(result)
[-0.3333 -0.3333  0.      1.667   0.6665]
```

mindspore.mint.nn.LogSigmoid

class mindspore.mint.nn.**LogSigmoid**

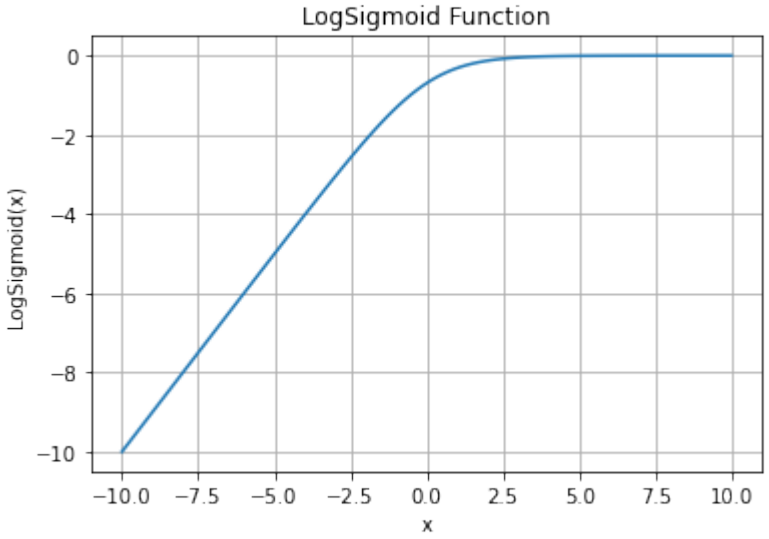
Applies logsigmoid activation element-wise. The input is a Tensor with any valid shape.

Logsigmoid is defined as:

$$\text{LogSigmoid}(x_i) = \log(\frac{1}{1 + \exp(-x_i)}),$$

where x_i is the element of the input.

LogSigmoid Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **input** (Tensor) - The input of LogSigmoid with data type of bfloat16, float16 or float32. The shape is $(*)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If dtype of *input* is not bfloat16, float16 and float32.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> net = mint.nn.LogSigmoid()
>>> input = Tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> output = net(input)
>>> print(output)
[-0.31326166 -0.12692806 -0.04858734]
```

mindspore.mint.nn.LogSoftmax

class mindspore.mint.nn.**LogSoftmax**(*dim=None*)

Applies the Log Softmax function to the input tensor on the specified axis. Supposes a slice in the given axis, x for each element x_i, the Log Softmax function is shown as follows:

$$\text{output}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)}\right),$$

where N is the length of the Tensor.

Parameters

dim (*int, optional*) – The axis to perform the Log softmax operation. Default: `None` .

Returns

Tensor, with the same shape as the input.

Raises

ValueError – If *dim* is not in range [-len(input.shape), len(input.shape)).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> log_softmax = mint.nn.LogSoftmax(dim=-1)
>>> output = log_softmax(x)
>>> print(output)
[[-5.00672150e+00 -6.72150636e-03 -1.20067215e+01]
 [-7.00091219e+00 -1.40009127e+01 -9.12250078e-04]]
```

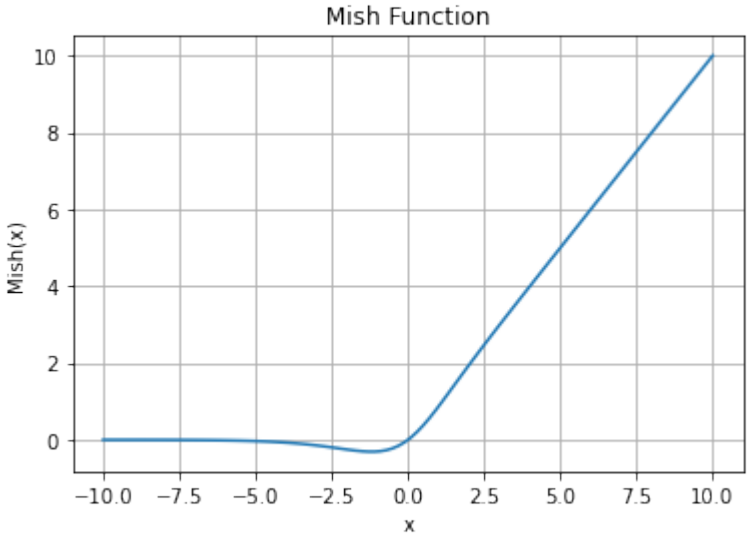
mindspore.mint.nn.Mish

class mindspore.mint.nn.**Mish**

Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.

Refer to *mindspore.mint.nn.functional.mish()* for more details.

Mish Activation Function Graph:



Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.array([[-1.1, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> mish = mint.nn.Mish()
>>> output = mish(x)
>>> print(output)
[[-3.0764845e-01 3.9974124e+00 -2.6832507e-03]
 [ 1.9439589e+00 -3.3576239e-02 8.9999990e+00]]
```

mindspore.mint.nn.PReLU

class mindspore.mint.nn.**PReLU**(*num_parameters=1*, *init=0.25*, *dtype=None*)

Applies PReLU activation function element-wise.

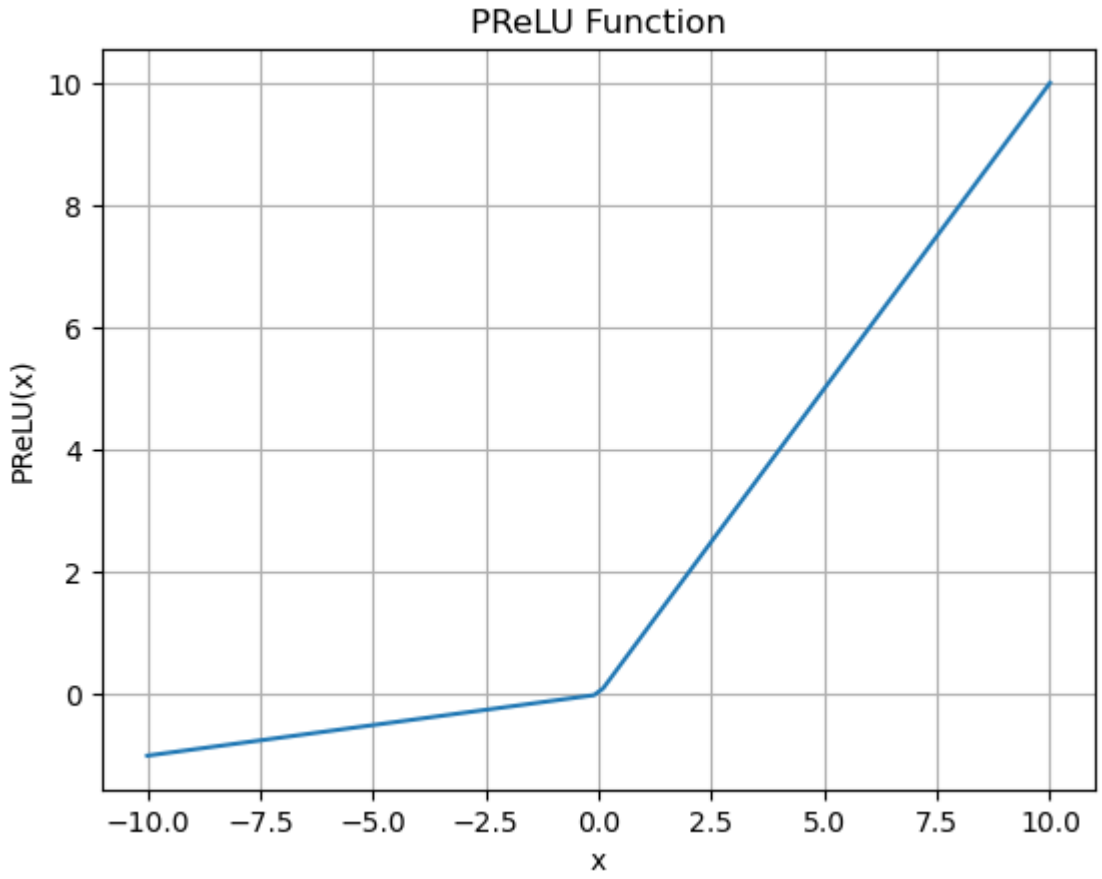
PReLU is defined as:

$$PReLU(x_i) = \max(0, x_i) + w * \min(0, x_i),$$

where x_i is an element of an channel of the input.

Here w is a learnable parameter with a default initial value 0.25. Parameter w has dimensionality of the argument channel. If called without argument channel, a single parameter w will be shared across all channels.

PReLU Activation Function Graph:



Note: Channel dim is the 2nd dim of input. When input has dims < 2, then there is no channel dim and the number of channels = 1.

Parameters

- **num_parameters** (*int, optional*) – number of *w* to learn. Although it takes an int as input, there is only two legitimate values: 1, or the number of channels at Tensor *input*. Default: `1` .
- **init** (*float, optional*) – the initial value of *w*. Default: `0.25` .
- **dtype** (`mindspore.dtype`, *optional*) – the type of *w*. Default: `None` . Supported data type is {float16, float32, bfloat16}.

Inputs:

- **input** (Tensor) - The input of PReLU.

Outputs:

Tensor, with the same dtype and shape as the *input*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.array([[[[0.1, 0.6], [0.9, 0.9]]]]), mindspore.float32)
>>> prelu = mint.nn.PReLU()
>>> output = prelu(x)
>>> print(output)
[[[[0.1 0.6]
   [0.9 0.9]]]]
```

mindspore.mint.nn.ReLU

class mindspore.mint.nn.**ReLU**

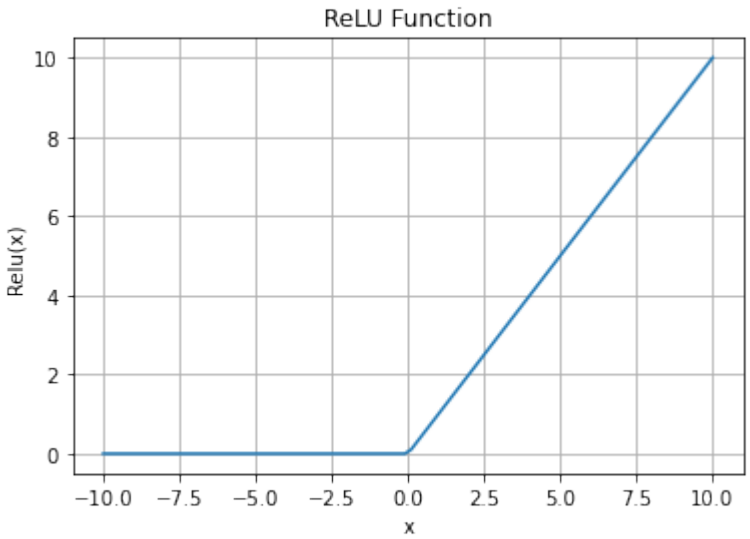
Applies ReLU (Rectified Linear Unit activation function) element-wise.

$$\text{ReLU}(input) = (input)^+ = \max(0, input),$$

It returns element-wise $\max(0, input)$.

Note: The neurons with the negative output will be suppressed and the active neurons will stay the same.

ReLU Activation Function Graph:



Inputs:

- **input** (Tensor) - The input of ReLU is a Tensor of any dimension.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is not supported.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> relu = mint.nn.ReLU()
>>> output = relu(input)
>>> print(output)
[0. 2. 0. 2. 0.]
```

mindspore.mint.nn.ReLU6

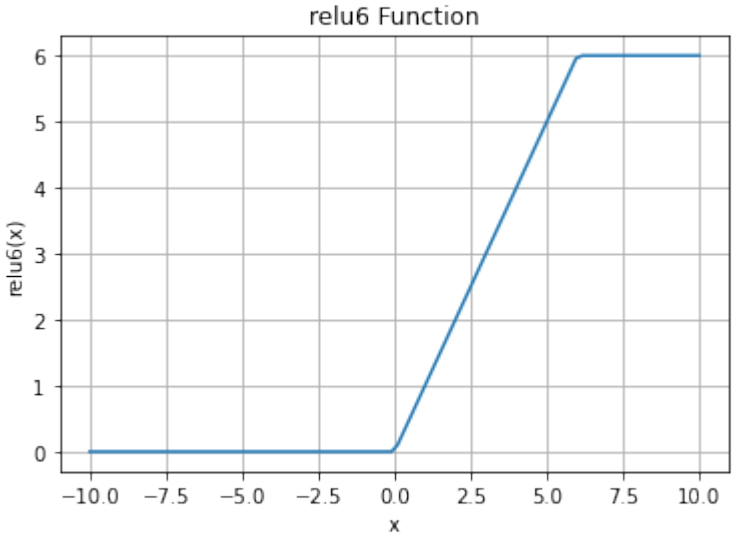
class mindspore.mint.nn.**ReLU6**(*inplace=False*)

Activation function ReLU6.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.mint.nn.functional.relu6()* for more details.

ReLU6 Activation Function Graph:



Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> relu6 = mint.nn.ReLU6()
>>> output = relu6(input)
>>> print(output)
[[0. 4. 0.]
[2. 0. 6.]]
```

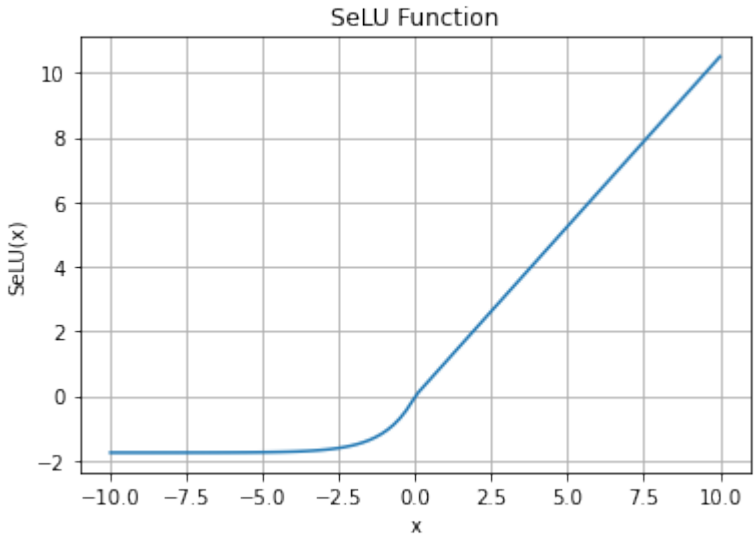
mindspore.mint.nn.SELU

class mindspore.mint.nn.**SELU**

Activation function SELU (Scaled exponential Linear Unit).

Refer to *mindspore.mint.nn.functional.selu()* for more details.

SELU Activation Function Graph:



Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> selu = mint.nn.SELU()
>>> output = selu(input)
>>> print(output)
[[-1.1113307 4.202804 -1.7575096]
[ 2.101402 -1.7462534 9.456309 ]]
```

mindspore.mint.nn.SiLU

class mindspore.mint.nn.**SiLU**

Calculates the SiLU activation function element-wise. It is also sometimes referred to as Swish function.

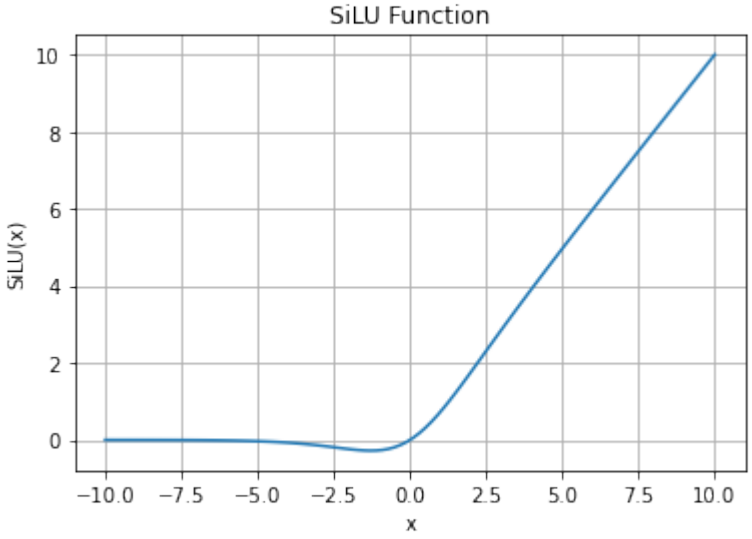
The SiLU function is defined as follows:

$$\text{SiLU}(x) = x * \sigma(x),$$

where x_i is an element of the input, $\sigma(x)$ is Sigmoid function.

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

SiLU Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **input** (Tensor) - *input* is x in the preceding formula. Input with the data type float16 or float32. Tensor of any dimension.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is neither float16 nor float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> silu = mint.nn.SiLU()
>>> output = silu(input)
>>> print(output)
[-0.269  1.762  -0.1423  1.762  -0.269]
```

mindspore.mint.nn.Sigmoid

class mindspore.mint.nn.**Sigmoid**

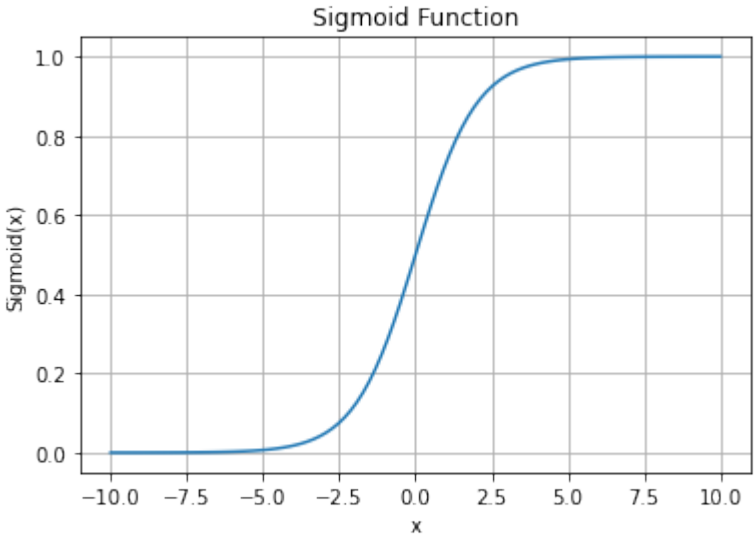
Applies sigmoid activation function element-wise.

Sigmoid function is defined as:

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

where x_i is the element of x.

Sigmoid Activation Function Graph:



Inputs:

- **input** (Tensor) - *input* is x in the preceding formula. Tensor of any dimension, the data type is float16, float32, float64, complex64 or complex128.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If dtype of *input* is not float16, float32, float64, complex64 or complex128.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> sigmoid = mint.nn.Sigmoid()
>>> output = sigmoid(input)
>>> print(output)
[0.2688  0.11914 0.5     0.881   0.7305 ]
```

mindspore.mint.nn.Softmax

class mindspore.mint.nn.**Softmax**(*dim=None*)

Applies the Softmax function to an n-dimensional input Tensor.

For details, please refer to *mindspore.mint.nn.functional.softmax()*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> softmax = mint.nn.Softmax()
>>> output = softmax(input)
>>> print(output)
[0.03168 0.01166 0.0861  0.636   0.2341 ]
```

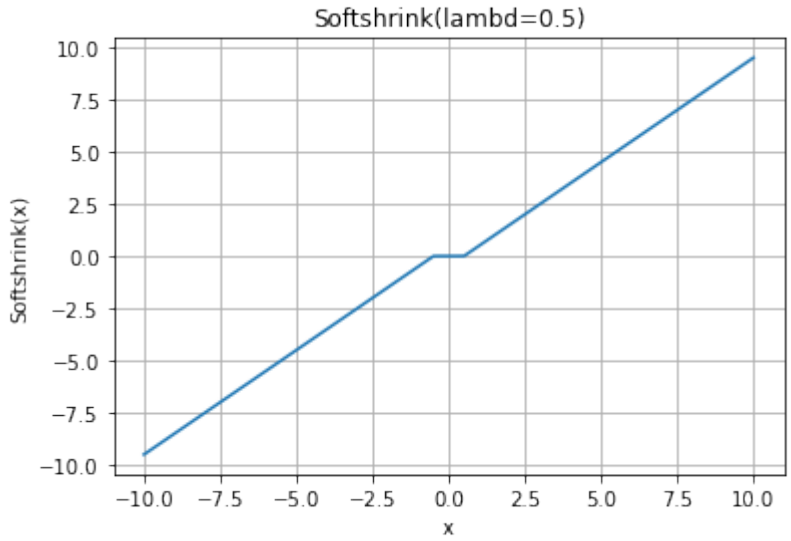
mindspore.mint.nn.Softshrink

class mindspore.mint.nn.**Softshrink**(*lambd=0.5*)

Applies the Softshrink function element-wise.

$$\text{Softshrink}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

Softshrink Activation Function Graph:



Parameters

lambd (*number, optional*) – The threshold λ defined by the Soft Shrink formula. It should be greater than or equal to 0, default: `0.5`.

Inputs:

- **input** (Tensor) - The input of Soft Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.

Outputs:

Tensor, the same shape and data type as the input.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[ 0.5297,  0.7871,  1.1754], [ 0.7836,  0.6218, -1.1542]]),␣
↪mindspore.float16)
>>> softshrink = mint.nn.Softshrink()
>>> output = softshrink(input)
>>> print(output)
[[ 0.02979  0.287    0.676  ]
 [ 0.2837   0.1216  -0.6543 ]]
```

mindspore.mint.nn.Tanh

class mindspore.mint.nn.**Tanh**

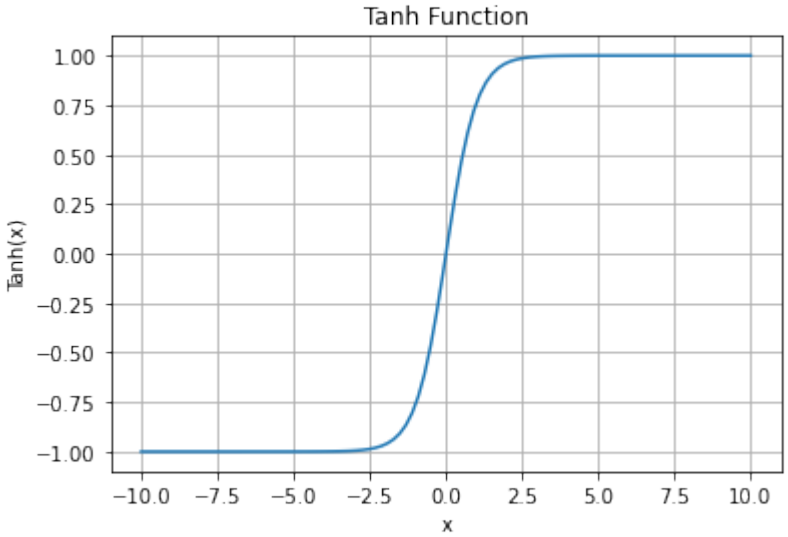
Applies the Tanh function element-wise, returns a new tensor with the hyperbolic tangent of the elements of input.

Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Tanh Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **input** (Tensor) - Tensor of any dimension, input with data type of float16 or float32.

Outputs:

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is neither float16 nor float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([1, 2, 3, 2, 1]), mindspore.float16)
>>> tanh = mint.nn.Tanh()
>>> output = tanh(input)
>>> print(output)
[0.7617 0.964  0.995  0.964  0.7617]
```

4.4.4 Embedding Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.Embedding	The value in *input* is used as the index, and the corresponding embedding vector is queried from *weight* .	Ascend

mindspore.mint.nn.Embedding

class mindspore.mint.nn.**Embedding**(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*, *_weight=None*, *_freeze=False*, *dtype=None*)

The value in *input* is used as the index, and the corresponding embedding vector is queried from *weight* .

Warning:

- This is an experimental API that is subject to change or deletion.
- On Ascend, the behavior is unpredictable when the value of *input* is invalid.

Parameters

- **num_embeddings** (*int*) –Size of the dictionary of embeddings.
- **embedding_dim** (*int*) –The size of each embedding vector.
- **padding_idx** (*int, optional*) –If the value is not None, the corresponding row of embedding vector will not be updated in training. The value of embedding vector at *padding_idx* will default to zeros when the Embedding layer is newly constructed. The value should be in range *[-num_embeddings, num_embeddings)* if it's not `None`. Default `None`.
- **max_norm** (*float, optional*) –If the value is not None, firstly get the p-norm result of the embedding vector specified by *input* where p is specified by *norm_type*; if the result is larger then *max_norm*, update the embedding vector with $\frac{max\_norm}{result+1e^{-7}}$. Default `None`.
- **norm_type** (*float, optional*) –Indicated the value of p in p-norm. Default `2.0`.
- **scale_grad_by_freq** (*bool, optional*) –If `True` the gradients will be scaled by the inverse of frequency of the index in *input*. Default `False`.
- **sparse** (*bool, optional*) –If `True`, gradient w.r.t. *weight* matrix will be a sparse tensor which has not been supported. Default: `False`.

- **_weight** (Tensor, *optional*) –Used to initialize the *weight* of Embedding. If None, the weight will be initialized from normal distribution $N(\text{sigma}=1.0, \text{mean}=0.0)$. Default None.
- **_freeze** (*bool*, *optional*) –If *weight* , the learnable weights of this module, should be freezed. Default: False.
- **dtype** (mindspore.dtype, *optional*) –Dtype of Embedding's *weight* . It is meaningless when *_weight* is not None. Default: None.

Variables:

- **weight** (Parameter) - The learnable weights of this module of shape (num_embeddings, embedding_dim), which initialized from $N(\text{sigma}=1.0, \text{mean}=0.0)$ or *_weight* .

Inputs:

- **input** (Tensor) - The indices used to lookup in the embedding vector. The data type must be int32 or int64, and the value should be in range *[0, num_embeddings)*.

Outputs:

Tensor, has the same data type as weight, the shape is $(*input.shape, embedding\_dim)$.

Raises

- **TypeError** –If *num_embeddings* is not an int.
- **TypeError** –If *embedding_dim* is not an int.
- **ValueError** –If *padding_idx* is out of valid range.
- **TypeError** –If *max_norm* is not a float.
- **TypeError** –If *norm_type* is not a float.
- **TypeError** –If *scale_grad_by_freq* is not a bool.
- **ValueError** –If *weight.shape* is invalid.
- **TypeError** –If *dtype* is not one of mindspore.dtype.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> mindspore.set_seed(0)
>>> input = Tensor([[1, 0, 1, 1], [0, 0, 1, 0]])
>>> embedding = mint.nn.Embedding(num_embeddings=10, embedding_dim=3)
>>> output = embedding(input)
>>> print(output)
[[[ 0.6712398   0.5407775   1.0317237]
  [-0.49091062 -0.42302188 -1.4807187]
  [ 0.6712398   0.5407775   1.0317237]
  [ 0.0024154   0.5407775   1.0317237]]
 [[-0.49091062 -0.42302188 -1.4807187]
  [-0.49091062 -0.42302188 -1.4807187]
  [ 0.6712398   0.5407775   1.0317237]
  [-0.49091062 -0.42302188 -1.4807187]]]
```

4.4.5 Linear Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.Linear	The linear connected layer.	Ascend

mindspore.mint.nn.Linear

class mindspore.mint.nn.**Linear**(*in_features*, *out_features*, *bias=True*, *weight_init=None*, *bias_init=None*, *dtype=None*)
The linear connected layer.

Applies linear connected layer for the input. This layer implements the operation as:

$$\text{outputs} = X * kernel + bias$$

Warning: On the Ascend platform, if *bias* is `False` , the *x* cannot be greater than 6D in PYNATIVE or KBK mode.

where X is the input tensors, kernel is a weight matrix with the same data type as the X created by the layer, and bias is a bias vector with the same data type as the X created by the layer (only if the parameter *bias* is True).

Warning: In PyNative mode, if *bias* is `False` , the *x* cannot be greater than 6D.

Parameters

- **in_features** (*int*) –The number of features in the input space.
- **out_features** (*int*) –The number of features in the output space.
- **bias** (*bool, optional*) –Specifies whether the layer uses a bias vector bias. Default: `True`.
- **weight_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable weight_init parameter. The dtype is same as *x*. The values of str refer to the function *initializer*. Default: `None` , weight will be initialized using HeUniform.
- **bias_init** (*Union[Tensor, str, Initializer, numbers.Number], optional*) –The trainable bias_init parameter. The dtype is same as *x*. The values of str refer to the function *initializer*. Default: `None` , bias will be initialized using Uniform.
- **dtype** (*mindspore.dtype*, optional) –Data type of Parameter. Default: `None` . If *dtype* is `None` , *dtype* is set to `mstype.float32` when initializing the method. When *weight_init* is Tensor, Parameter has the same data type as *weight_init* , in other cases, Parameter has the same data type as *dtype*, the same goes for *bias_init*.

Inputs:

- **x** (Tensor) - Tensor of shape $(*, in\_features)$. The *in_features* in *Args* should be equal to $in\_features$ in *Inputs*.

Outputs:

Tensor of shape $(*, out\_features)$.

Raises

- **TypeError** –If *in_features* or *out_features* is not an int.
- **TypeError** –If *bias* is not a bool.

- **ValueError** –If length of shape of *weight_init* is not equal to 2 or shape[0] of *weight_init* is not equal to *out_features* or shape[1] of *weight_init* is not equal to *in_features*.
- **ValueError** –If length of shape of *bias_init* is not equal to 1 or shape[0] of *bias_init* is not equal to *out_features*.
- **RuntimeError** –On the Ascend platform, if *bias* is `False` and *x* is greater than 6D in PYNATIVE or KBK mode.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> import numpy as np
>>> x = Tensor(np.array([[180, 234, 154], [244, 48, 247]]), mindspore.float32)
>>> net = mint.nn.Linear(3, 4)
>>> output = net(x)
>>> print(output.shape)
(2, 4)
```

4.4.6 Dropout Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.Dropout	Dropout layer for the input.	`Ascend`
mindspore.mint.nn.Dropout2d	During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of *NCHW*, the channel feature map refers to a 2-dimensional feature map with the shape of *HW*).	`Ascend`

mindspore.mint.nn.Dropout

class mindspore.mint.nn.**Dropout**(*p=0.5*, *inplace=False*)

Dropout layer for the input.

Dropout is a means of regularization that reduces overfitting by preventing correlations between neuronal nodes. The operator randomly sets some neurons output to 0 according to *p*, which means the probability of discarding during training. And the return will be multiplied by $\frac{1}{1-p}$ during training. During the reasoning, this layer returns the same Tensor as the *x*.

This technique is proposed in paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting and proved to be effective to reduce over-fitting and prevents neurons from co-adaptation. See more details in Improving neural networks by preventing co-adaptation of feature detectors.

Note:

- Each channel will be zeroed out independently on every construct call.
- Parameter *p* means the probability of the element of the input tensor to be zeroed.

Parameters

- **p** (*float, optional*) –The dropout rate of input neurons, E.g. *p* =0.9, dropping out 90% of input neurons. Default: `0.5` .
- **inplace** (*bool, optional*) –Whether to enable the operation in-place. If set to `True` , will do this operation in-place. Default: `False` .

Inputs:

- **x** (Tensor) - The input of Dropout.

Outputs:

Tensor, output tensor with the same shape as the *x*.

Raises

- **TypeError** –If the dtype of *p* is not float.
- **ValueError** –If length of shape of *x* is less than 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.ones([2, 2, 3]), mindspore.float32)
>>> net = mint.nn.Dropout(p=0.2)
>>> net.set_train()
>>> output = net(x)
>>> print(output.shape)
(2, 2, 3)
```

mindspore.mint.nn.Dropout2d

class mindspore.mint.nn.**Dropout2d**(*p=0.5*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of *NCHW*, the channel feature map refers to a 2-dimensional feature map with the shape of *HW*).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *2D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call with probability *p* using samples from a Bernoulli distribution.

Dropout2d can improve the independence between channel feature maps.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.mint.nn.functional.dropout2d()* for more details.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> dropout = mint.nn.Dropout2d(p=0.5)
>>> x = Tensor(np.ones([2, 1, 2, 3]), mindspore.float32)
>>> output = dropout(x)
>>> print(output.shape)
(2, 1, 2, 3)
```

4.4.7 Pooling Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.AdaptiveAvgPool1d	Applies a 1D adaptive average pooling over an input signal composed of several input planes.	Ascend
mindspore.mint.nn.AdaptiveAvgPool2d	Applies a 2D adaptive average pooling over an input signal composed of several input planes.	Ascend
mindspore.mint.nn.AdaptiveAvgPool3d	This operator applies a 3D adaptive average pooling to an input signal composed of multiple input planes.	Ascend
mindspore.mint.nn.AdaptiveMaxPool1d	Applies a 1D adaptive max pooling over an input signal composed of several input planes.	Ascend
mindspore.mint.nn.AvgPool2d	Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.	Ascend
mindspore.mint.nn.AvgPool3d	Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes.	Ascend
mindspore.mint.nn.MaxUnpool2d	Computes the inverse of *Maxpool2d*.	Ascend

mindspore.mint.nn.AdaptiveAvgPool1d

class mindspore.mint.nn.**AdaptiveAvgPool1d**(*output_size*)

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

The output is of size L_{out} , for any input size. The number of output features is equal to the number of input planes.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

output_size (*int*) – the target output size L_{out} .

Inputs:

- **input** (Tensor) - The input with shape (N, C, L_{in}) or (C, L_{in}) .

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[[2, 1, 2], [2, 3, 5]]]), mindspore.float16)
>>> net = mint.nn.AdaptiveAvgPool1d(3)
>>> output = net(input)
>>> print(output)
[[[2. 1. 2.]
  [2. 3. 5.]]]
```

mindspore.mint.nn.AdaptiveAvgPool2d

class mindspore.mint.nn.**AdaptiveAvgPool2d**(*output_size*)

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

The output is of size HxW , for any input size. The number of output features is equal to the number of input planes.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

output_size (*Union(int, tuple[int])*) – the target output size of the image of the form HxW . Can be a tuple (H, W) or a single H for square image HxH . H and W can be either a `int` , or `None` which means the size will be the same as that of the input.

Inputs:

- **input** (Tensor) - The input with shape (N, C, H, W) or (C, H, W) .

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[[2, 1, 2], [2, 3, 5]]]), mindspore.float16)
>>> net = mint.nn.AdaptiveAvgPool2d((2, 2))
>>> output = net(input)
>>> print(output)
[[[1.5 1.5]
  [2.5 4. ]]]
```

mindspore.mint.nn.AdaptiveAvgPool3d

class mindspore.mint.nn.**AdaptiveAvgPool3d**(*output_size*)

This operator applies a 3D adaptive average pooling to an input signal composed of multiple input planes. That is, for any input size, the size of the specified output is (D, H, W). The number of output features is equal to the number of input planes.

Suppose the last 3 dimension size of input is (inD, inH, inW), then the last 3 dimension size of output is $(outD, outH, outW)$.

$$
\begin{aligned}
&\forall \quad od \in [0, outD-1], oh \in [0, outH-1], ow \in [0, outW-1] \\
&output[od, oh, ow] = \\
&\qquad mean(input[istartD : iendD+1, istartH : iendH+1, istartW : iendW+1]) \\
&where, \\
&\qquad istartD = \left\lceil \frac{od * inD}{outD} \right\rceil \\
&\qquad iendD = \left\lfloor \frac{(od+1) * inD}{outD} \right\rfloor \\
&\qquad istartH = \left\lceil \frac{oh * inH}{outH} \right\rceil \\
&\qquad iendH = \left\lfloor \frac{(oh+1) * inH}{outH} \right\rfloor \\
&\qquad istartW = \left\lceil \frac{ow * inW}{outW} \right\rceil \\
&\qquad iendW = \left\lfloor \frac{(ow+1) * inW}{outW} \right\rfloor
\end{aligned}
$$

> **Warning:** For Ascend, it is only supported on Atlas A2 Training Series Products. This is an experimental optimizer API that is subject to change or deletion.

Parameters

output_size (*Union[int, tuple]*) – The target output size. *output_size* can be a tuple (D, H, W), or an int D for (D, D, D). D, H and W can be int or None which means the output size is the same as that of the input.

Inputs:

- **input** (Tensor) - The input of AdaptiveAvgPool3d, which is a 5D or 4D Tensor.

Outputs:

Tensor, with the same type as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **ValueError** – If the dimension of *input* is not 4D or 5D.
- **ValueError** – If *output_size* value is not positive.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> # case 1: output_size=(3, 3, 4)
>>> output_size=(3, 3, 4)
>>> input_x_val = np.random.randn(4, 3, 5, 6, 7)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = mint.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(4, 3, 3, 3, 4)
>>> # case 2: output_size=4
>>> output_size=5
>>> input_x_val = np.random.randn(2, 3, 8, 6, 12)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = mint.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(2, 3, 5, 5, 5)
>>> # case 3: output_size=(None, 4, 5)
>>> output_size=(None, 4, 5)
>>> input_x_val = np.random.randn(4, 1, 9, 10, 8)
>>> input_x = ms.Tensor(input_x_val, ms.float32)
>>> net = mint.nn.AdaptiveAvgPool3d(output_size)
>>> output = net(input_x)
>>> print(output.shape)
(4, 1, 9, 4, 5)
```

mindspore.mint.nn.AdaptiveMaxPool1d

class mindspore.mint.nn.**AdaptiveMaxPool1d**(*output_size*, *return_indices=False*)

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

The output is of size L_{out} , for any input size. The number of output features is equal to the number of input planes.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **output_size** (*Union[int, tuple]*) –the target output size L_{out} .
- **return_indices** (*bool, optional*) –Whether to return the index of the maximum value. Default: `False` .

Inputs:

- **input** (Tensor) - The input with shape (N, C, L_{in}) or (C, L_{in}) .

Outputs:

Union(Tensor, tuple(Tensor, Tensor)).

- If *return_indices* is False, output is a Tensor, with shape (N, C, L_{out}). It has the same data type as x.
- If *return_indices* is True, output is a Tuple of 2 Tensors, representing the result and where the max values are generated.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or float64.
- **TypeError** – If *output_size* is not int or tuple.
- **TypeError** – If *return_indices* is not a bool.
- **ValueError** – If *output_size* is a tuple and the length of *output_size* is not 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[[2, 1, 2], [2, 3, 5]]]), mindspore.float16)
>>> net = mint.nn.AdaptiveMaxPool1d(3)
>>> output = net(input)
>>> print(output)
[[[2. 1. 2.]
  [2. 3. 5.]]]
```

mindspore.mint.nn.AvgPool2d

class mindspore.mint.nn.**AvgPool2d**(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*)

Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.

For details, please refer to *mindspore.mint.nn.functional.avg_pool2d()*.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> input = Tensor(np.arange(1 * 3 * 3 * 4).reshape(1, 3, 3, 4), mstype.float32)
>>> net = mint.nn.AvgPool2d(kernel_size=2, stride=1)
>>> output = net(input)
>>> print(output.shape)
(1, 3, 2, 3)
```

mindspore.mint.nn.AvgPool3d

class mindspore.mint.nn.**AvgPool3d**(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*)

Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes.

> **Warning:** This is an experimental API that is subject to change or deletion.

For details, please refer to *mindspore.mint.nn.functional.avg_pool3d()*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> pool = ms.mint.nn.AvgPool3d(kernel_size=3, stride=1)
>>> x = ms.ops.randn(1, 2, 4, 4, 5).astype(ms.float32)
>>> output = pool(x)
>>> print(output.shape)
(1, 2, 2, 2, 3)
>>> x1 = ms.ops.randn(6, 5, 7, 7, 5).astype(ms.float32)
>>> pool2 = ms.mint.nn.AvgPool3d(4, stride=2, padding=(2, 2, 1), divisor_override=10)
>>> output2 = pool2(x1)
>>> print(output2.shape)
(6, 5, 4, 4, 2)
```

mindspore.mint.nn.MaxUnpool2d

class mindspore.mint.nn.**MaxUnpool2d**(*kernel_size*, *stride=None*, *padding=0*)

Computes the inverse of *Maxpool2d*.

MaxUnpool2d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}), and the output is of shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}). The operation is as follows.

$$H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$
$$W_{out} = (W_{in} - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1]$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]], optional*) – The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: `None` , which indicates the moving step is *kernel_size* .

- **padding** (*Union[int, tuple[int]], optional*) –The pad value to be filled. Default: `0` . If *padding* is an integer, the paddings of height and width are the same, equal to padding. If *padding* is a tuple of two integers, the padding of height and width equal to padding[0] and padding[1] correspondingly.

Inputs:

- **input** (Tensor) - The input Tensor to invert. Tensor of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- **indices** (Tensor) - Max values' index represented by the indices. Tensor of shape must be same with input 'input'. Values of indices must belong to $[0, H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **output_size** (tuple[int], optional) - The target output size. Default: `None` . If output_size == (), then the shape of output computed by *kernel_size*, *stride* and *padding*. If output_size != (), then output_size must be (N, C, H, W) , (C, H, W) or (H, W) and output_size must belong to $[(N, C, H_{out} - stride[0], W_{out} - stride[1]), (N, C, H_{out} + stride[0], W_{out} + stride[1])]$.

Outputs:

Tensor, with shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), with the same data type with *input*.

Raises

- **TypeError** –If data type of *input* or *indices* is not supported.
- **TypeError** –If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** –If numbers in *stride*, *padding* or *kernel_size* is not positive.
- **ValueError** –If the shapes of *input* and *indices* are not equal.
- **ValueError** –If *input* whose length is not 3 or 4.
- **ValueError** –If *output_size* whose type is not tuple.
- **ValueError** –If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[[[0, 1], [8, 9]]]]).astype(np.float32))
>>> indices = Tensor(np.array([[[[0, 1], [2, 3]]]]).astype(np.int64))
>>> net =  mint.nn.MaxUnpool2d(1, stride=1, padding=0)
>>> output = net(input, indices)
>>> print(output.asnumpy())
[[[[0. 1.]
   [8. 9.]]]]
```

4.4.8 Padding Layers

API Name	Description	Supported Platforms
mindspore.mint.nn.ConstantPad1d	Pad the last dimension of *input* tensor using *padding* and *value*.	Ascend
mindspore.mint.nn.ConstantPad2d	Pad the last 2 dimensions of *input* tensor using *padding* and *value*.	Ascend
mindspore.mint.nn.ConstantPad3d	Pad the last 3 dimension of *input* tensor using *padding* and *value*.	Ascend
mindspore.mint.nn.ReflectionPad1d	Pad the last dimension of *input* tensor using the reflection of the input boundary.	Ascend
mindspore.mint.nn.ReflectionPad2d	Pad the last 2 dimension of *input* tensor using the reflection of the input boundary.	Ascend
mindspore.mint.nn.ReflectionPad3d	Pad the last 3 dimension of *input* tensor using the reflection of the input boundary.	Ascend
mindspore.mint.nn.ReplicationPad1d	Pad the last dimension of *input* tensor using the replication of the input boundary.	Ascend
mindspore.mint.nn.ReplicationPad2d	Pad the last 2 dimension of *input* tensor using the replication of the input boundary.	Ascend
mindspore.mint.nn.ReplicationPad3d	Pad the last 3 dimension of *input* tensor using the replication of the input boundary.	Ascend
mindspore.mint.nn.ZeroPad1d	Pad the last dimension of *input* tensor with 0 using *padding*.	Ascend
mindspore.mint.nn.ZeroPad2d	Pad the last 2 dimension of *input* tensor with 0 using *padding*.	Ascend
mindspore.mint.nn.ZeroPad3d	Pad the last 3 dimension of *input* tensor with 0 using *padding*.	Ascend

mindspore.mint.nn.ConstantPad1d

class mindspore.mint.nn.**ConstantPad1d**(*padding*, *value*)

Pad the last dimension of *input* tensor using *padding* and *value*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **padding** (*Union[int, tuple, list]*) –Specifies padding size.
- **value** (*Union[int, float]*) –Specifies padding value.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 2 integers.
- **TypeError** – If *input* is not Tensor.
- **TypeError** – If *value* is not int or float.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> # padding is tuple
>>> padding = (0, 1)
>>> value = 0.5
>>> pad1d = ms.mint.nn.ConstantPad1d(padding, value)
>>> out = pad1d(x)
>>> print(out)
[[[[1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]]
  [[1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]
   [1.  1.  1.  1.  0.5]]]]
>>> print(out.shape)
(1, 2, 3, 5)
>>> # padding is int
>>> padding = 1
>>> value = 0.5
>>> pad1d = ms.mint.nn.ConstantPad1d(padding, value)
>>> out = pad1d(x)
>>> print(out)
[[[[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]]
  [[0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]
   [0.5 1.  1.  1.  1.  0.5]]]]
>>> print(out.shape)
(1, 2, 3, 6)
```

mindspore.mint.nn.ConstantPad2d

class mindspore.mint.nn.**ConstantPad2d**(*padding*, *value*)

Pad the last 2 dimensions of *input* tensor using *padding* and *value*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **padding** (*Union[int, tuple, list]*) –Specifies padding size.
- **value** (*Union[int, float]*) –Specifies padding value.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** –If *padding* is not an integer of a list or tuple of 4 integers.
- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *value* is not int or float.
- **ValueError** –If *padding* contains negative value.
- **ValueError** –If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1)
>>> value = 0.5
>>> pad2d = ms.mint.nn.ConstantPad2d(padding, value)
>>> out = pad2d(x)
>>> print(out)
[[[[0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  0.5 0.5 0.5 0.5 0.5]]
  [[0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  1.  1.  1.  1.  0.5]
   [0.5  0.5 0.5 0.5 0.5 0.5]]]]
```

(continues on next page)

(continued from previous page)

```
>>> print(out.shape)
(1, 2, 4, 6)
```

mindspore.mint.nn.ConstantPad3d

class mindspore.mint.nn.**ConstantPad3d**(*padding*, *value*)

Pad the last 3 dimension of *input* tensor using *padding* and *value*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **padding** (*Union[int, tuple, list]*) –Specifies padding size.
- **value** (*Union[int, float]*) –Specifies padding value.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** –If *padding* is not an integer of a list or tuple of 6 integers.
- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *value* is not int or float.
- **ValueError** –If *padding* contains negative value.
- **ValueError** –If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1, 1, 0)
>>> value = 0.5
>>> pad3d = ms.mint.nn.ConstantPad3d(padding, value)
>>> out = pad3d(x)
>>> print(out)
[[[[0.5 0.5 0.5 0.5 0.5 0.5]
   [0.5 0.5 0.5 0.5 0.5 0.5]
```

(continues on next page)

(continued from previous page)

```
    [0.5 0.5 0.5 0.5 0.5 0.5]
    [0.5 0.5 0.5 0.5 0.5 0.5]]
   [[0.5 1.  1.  1.  1.  0.5]
    [0.5 1.  1.  1.  1.  0.5]
    [0.5 1.  1.  1.  1.  0.5]
    [0.5 0.5 0.5 0.5 0.5 0.5]]
   [[0.5 1.  1.  1.  1.  0.5]
    [0.5 1.  1.  1.  1.  0.5]
    [0.5 1.  1.  1.  1.  0.5]
    [0.5 0.5 0.5 0.5 0.5 0.5]]]]
>>> print(out.shape)
(1, 3, 4, 6)
```

mindspore.mint.nn.ReflectionPad1d

class mindspore.mint.nn.**ReflectionPad1d**(*padding*)

Pad the last dimension of *input* tensor using the reflection of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 2D or 3D input Tensor with shape: (C, W_{in}) or (N, C, W_{in}).

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 2 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = ms.Tensor(np.array([[[0, 1, 2, 3], [4, 5, 6, 7]]]).astype(np.float32))
>>> # x has shape (1, 2, 4)
>>> padding = (3, 1)
>>> # The first and the second dimension of x remain the same.
>>> # The third dimension of x: W_out = W_in + pad_left + pad_right = 4 + 3 + 1 = 8
>>> pad1d = ms.mint.nn.ReflectionPad1d(padding)
>>> out = pad1d(x)
>>> # The shape of out is (1, 2, 8)
>>> print(out)
[[[3. 2. 1. 0. 1. 2. 3. 2.]
  [7. 6. 5. 4. 5. 6. 7. 6.]]]
```

mindspore.mint.nn.ReflectionPad2d

class mindspore.mint.nn.**ReflectionPad2d**(*padding*)

Pad the last 2 dimension of *input* tensor using the reflection of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 3D or 4D input Tensor with shape: (C, H_{in}, W_{in}) or (N, C, H_{in}, W_{in}).

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 4 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = ms.Tensor(np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]]]).astype(np.float32))
>>> # x has shape (1, 3, 3)
>>> padding = (1, 1, 2, 0)
>>> pad2d = ms.mint.nn.ReflectionPad2d(padding)
>>> # The first dimension of x remains the same.
>>> # The second dimension of x: H_out = H_in + pad_up + pad_down = 3 + 1 + 1 = 5
>>> # The third dimension of x: W_out = W_in + pad_left + pad_right = 3 + 2 + 0 = 5
>>> out = pad2d(x)
>>> # The shape of out is (1, 5, 5)
>>> print(out)
[[[7. 6. 7. 8. 7.]
  [4. 3. 4. 5. 4.]
  [1. 0. 1. 2. 1.]
  [4. 3. 4. 5. 4.]
  [7. 6. 7. 8. 7.]]]
```

mindspore.mint.nn.ReflectionPad3d

class mindspore.mint.nn.**ReflectionPad3d**(*padding*)

Pad the last 3 dimension of *input* tensor using the reflection of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 4D or 5D input Tensor with shape: $(N, D_{in}, H_{in}, W_{in})$ or $(N, C, D_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 6 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> arr = np.arange(8).astype(np.float32).reshape((1, 2, 2, 2))
>>> x = ms.Tensor(arr)
>>> # x has shape (1, 2, 2, 2)
>>> padding = (1, 1, 1, 0, 0, 1)
>>> pad3d = ms.mint.nn.ReflectionPad3d(padding)
>>> out = pad3d(x)
>>> # The first dimension of x remains the same.
>>> # The second dimension of x: D_out = D_in + pad_front + pad_back = 2 + 0 + 1 = 3
>>> # The third dimension of x: H_out = H_in + pad_up + pad_down = 2 + 1 + 0 = 3
>>> # The last dimension of x: W_out = W_in + pad_left + pad_right = 2 + 1 + 1 = 4
>>> # The shape of out is (1, 3, 3, 4)
>>> print(out)
[[[[3. 2. 3. 2.]
   [1. 0. 1. 0.]
   [3. 2. 3. 2.]]
  [[7. 6. 7. 6.]
   [5. 4. 5. 4.]
   [7. 6. 7. 6.]]
  [[3. 2. 3. 2.]
   [1. 0. 1. 0.]
   [3. 2. 3. 2.]]]]
```

mindspore.mint.nn.ReplicationPad1d

class mindspore.mint.nn.**ReplicationPad1d**(*padding*)

Pad the last dimension of *input* tensor using the replication of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 2D or 3D input Tensor with shape: (C, W_{in}) or (N, C, W_{in}).

Outputs:

The tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 2 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad1d = ms.mint.nn.ReplicationPad1d(2)
>>> input = ms.Tensor(np.arange(0, 8).reshape(1, 2, 4), ms.float32)
>>> print(input)
[[[0. 1. 2. 3.]
  [4. 5. 6. 7.]]]
>>> out = pad1d(input)
>>> print(out)
[[[0. 0. 0. 1. 2. 3. 3. 3.]
  [4. 4. 4. 5. 6. 7. 7. 7.]]]
>>> pad1d = ms.mint.nn.ReplicationPad1d((3, 1))
>>> out = pad1d(input)
>>> print(out)
[[[0. 0. 0. 0. 1. 2. 3. 3.]
  [4. 4. 4. 4. 5. 6. 7. 7.]]]
```

mindspore.mint.nn.ReplicationPad2d

class mindspore.mint.nn.**ReplicationPad2d**(*padding*)

Pad the last 2 dimension of *input* tensor using the replication of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 3D or 4D input Tensor with shape: (C, H_{in}, W_{in}) or (N, C, H_{in}, W_{in}).

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 4 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad2d = ms.mint.nn.ReplicationPad2d(2)
>>> input = ms.Tensor(np.arange(0, 9).reshape(1, 1, 3, 3), ms.float32)
>>> print(input)
[[[[0. 1. 2.]
   [3. 4. 5.]
   [6. 7. 8.]]]]
>>> out = pad2d(input)
>>> print(out)
[[[[0. 0. 0. 1. 2. 2. 2.]
   [0. 0. 0. 1. 2. 2. 2.]
   [0. 0. 0. 1. 2. 2. 2.]
   [3. 3. 3. 4. 5. 5. 5.]
   [6. 6. 6. 7. 8. 8. 8.]
   [6. 6. 6. 7. 8. 8. 8.]
   [6. 6. 6. 7. 8. 8. 8.]]]]
>>> pad2d = ms.mint.nn.ReplicationPad2d((1, 1, 2, 0))
>>> out = pad2d(input)
>>> print(out)
[[[[0. 0. 1. 2. 2.]
   [0. 0. 1. 2. 2.]
   [0. 0. 1. 2. 2.]
   [3. 3. 4. 5. 5.]
   [6. 6. 7. 8. 8.]]]]
```

mindspore.mint.nn.ReplicationPad3d

class mindspore.mint.nn.**ReplicationPad3d**(*padding*)

Pad the last 3 dimension of *input* tensor using the replication of the input boundary.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - 4D or 5D input Tensor with shape: $(N, D_{in}, H_{in}, W_{in})$ or $(N, C, D_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 6 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> pad3d = ms.mint.nn.ReplicationPad3d(1)
>>> input = ms.Tensor(np.arange(0, 9).reshape(1, 1, 1, 3, 3), ms.float32)
>>> out = pad3d(input)
>>> print(out)
[[[[[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]
   [[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]
   [[0. 0. 1. 2. 2.]
    [0. 0. 1. 2. 2.]
    [3. 3. 4. 5. 5.]
    [6. 6. 7. 8. 8.]
    [6. 6. 7. 8. 8.]]]]]
```

mindspore.mint.nn.ZeroPad1d

class mindspore.mint.nn.**ZeroPad1d**(*padding*)

Pad the last dimension of *input* tensor with 0 using *padding*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 2 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.

- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> # padding is tuple
>>> padding = (0, 1)
>>> pad1d = ms.mint.nn.ZeroPad1d(padding)
>>> out = pad1d(x)
>>> print(out)
[[[[1.  1.  1.  1.  0.]
   [1.  1.  1.  1.  0.]
   [1.  1.  1.  1.  0.]]
  [[1.  1.  1.  1.  0.]
   [1.  1.  1.  1.  0.]
   [1.  1.  1.  1.  0.]]]]
>>> print(out.shape)
(1, 2, 3, 5)
>>> # padding is int
>>> padding = 1
>>> pad1d = ms.mint.nn.ZeroPad1d(padding)
>>> out = pad1d(x)
>>> print(out)
[[[[0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]]
  [[0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]]]]
>>> print(out.shape)
(1, 2, 3, 6)
```

mindspore.mint.nn.ZeroPad2d

class mindspore.mint.nn.**ZeroPad2d**(*padding*)

Pad the last 2 dimension of *input* tensor with 0 using *padding*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 4 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1)
>>> pad = ms.mint.nn.ZeroPad2d(padding)
>>> out = pad(x)
>>> print(out)
[[[[0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 0. 0. 0. 0. 0.]]
  [[0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 1. 1. 1. 1. 0.]
   [0. 0. 0. 0. 0. 0.]]]]
>>> print(out.shape)
(1, 2, 4, 6)
```

mindspore.mint.nn.ZeroPad3d

class mindspore.mint.nn.**ZeroPad3d**(*padding*)

Pad the last 3 dimension of *input* tensor with 0 using *padding*.

For more information, please refer to *mindspore.mint.nn.functional.pad()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

padding (*Union[int, tuple, list]*) – Specifies padding size.

Inputs:

- **input** (Tensor) - shape is $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *padding* is not an integer of a list or tuple of 6 integers.
- **TypeError** – If *input* is not Tensor.
- **ValueError** – If *padding* contains negative value.
- **ValueError** – If *padding* is a tuple or list, and the length does not match the tensor dimension.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> x = np.ones(shape=(1, 2, 3, 4)).astype(np.float32)
>>> x = ms.Tensor(x)
>>> padding = (1, 1, 0, 1, 1, 0)
>>> pad3d = ms.mint.nn.ZeroPad3d(padding)
>>> out = pad3d(x)
>>> print(out)
[[[[0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]]
  [[0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 0. 0. 0. 0. 0.]]
  [[0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 1.  1.  1.  1.  0.]
   [0. 0. 0. 0. 0. 0.]]]]
>>> print(out.shape)
(1, 3, 4, 6)
```

4.4.9 Loss Functions

API Name	Description	Supported Platforms
mindspore.mint.nn.BCELoss	Compute the binary cross entropy between the true labels and predicted labels.	`Ascend`
mindspore.mint.nn.BCEWithLogitsLoss	Adds sigmoid activation function to *input* as logits, and uses this logits to compute binary cross entropy between the logits and the target.	`Ascend`
mindspore.mint.nn.CrossEntropyLoss	The cross entropy loss between input and target.	`Ascend`
mindspore.mint.nn.KLDivLoss	Computes the Kullback-Leibler divergence between the *input* and the *target*.	`Ascend`

continues on next page

Table 17 – continued from previous page

mindspore.mint.nn.L1Loss	L1Loss is used to calculate the mean absolute error between the predicted value and the target value.	Ascend
mindspore.mint.nn.MSELoss	Calculates the mean squared error between the predicted value and the label value.	Ascend
mindspore.mint.nn.NLLLoss	Gets the negative log likelihood loss between inputs and target.	Ascend
mindspore.mint.nn.SmoothL1Loss	Computes smooth L1 loss, a robust L1 loss.	Ascend

mindspore.mint.nn.BCELoss

class mindspore.mint.nn.**BCELoss**(*weight=None*, *reduction='mean'*)

Compute the binary cross entropy between the true labels and predicted labels.

Set the predicted labels as x, true labels as y, the output loss as $\ell(x, y)$. The formula is as follow:

$$L = \{l_1, \ldots, l_n, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

where N is the batch size. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Note: Note that the predicted labels should always be the output of sigmoid. Because it is a two-class classification, the true labels should be numbers between 0 and 1. And if x_n is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **weight** (Tensor, *optional*) – A rescaling weight applied to the loss of each batch element. And it must have the same shape and data type as *inputs*. Default: None .
- **reduction** (*str*, *optional*) – Apply specific reduction method to the output: 'none' , 'mean' , 'sum' . Default: 'mean' .
  - 'none': no reduction will be applied.
  - 'mean': compute and return the weighted mean of elements in the output.
  - 'sum': the output elements will be summed.

Inputs:

- **input** (Tensor) - The input tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions. The data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products).
- **target** (Tensor) - The label tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions. The same shape and data type as *input*.

Outputs:

Tensor, has the same dtype as *input*. if *reduction* is 'none', then it has the same shape as *input*. Otherwise, it is a scalar Tensor.

Raises

- **TypeError** –If dtype of *input*, *target* or *weight* (if given) is not float16, float32 or bfloat16.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **ValueError** –If shape of *input* is not the same as *target* or *weight* (if given).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> weight = ms.Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 3.3, 2.2]]), ms.float32)
>>> loss = mint.nn.BCELoss(weight=weight, reduction='mean')
>>> input = ms.Tensor(np.array([[0.1, 0.2, 0.3], [0.5, 0.7, 0.9]]), ms.float32)
>>> target = ms.Tensor(np.array([[0, 1, 0], [0, 0, 1]]), ms.float32)
>>> output = loss(input, target)
>>> print(output)
1.8952923
```

mindspore.mint.nn.BCEWithLogitsLoss

class mindspore.mint.nn.**BCEWithLogitsLoss**(*weight=None*, *reduction='mean'*, *pos_weight=None*)

Adds sigmoid activation function to *input* as logits, and uses this logits to compute binary cross entropy between the logits and the target.

Sets input *input* as X, input *target* as Y, output as L. Then,

$$p_{ij} = sigmoid(X_{ij}) = \frac{1}{1 + e^{-X_{ij}}}$$

$$L_{ij} = -[Y_{ij} \cdot \log(p_{ij}) + (1 - Y_{ij}) \cdot \log(1 - p_{ij})]$$

Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **weight** (*Tensor, optional*) –A rescaling weight applied to the loss of each batch element. If not None, it can be broadcast to a tensor with shape of *target*, data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

- **pos_weight** (Tensor, *optional*) –A weight of positive examples. Must be a vector with length equal to the number of classes. If not None, it must be broadcast to a tensor with shape of *input*, data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None` .

Inputs:

- **input** (Tensor) - Input *input* with shape $(N, *)$ where $*$ means, any number of additional dimensions. The data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **target** (Tensor) - Ground truth label with shape $(N, *)$ where $*$ means, any number of additional dimensions. The same shape and data type as *input*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, its shape is the same as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If input *input* or *target* is not Tensor.
- **TypeError** –If *weight* or *pos_weight* is a parameter.
- **TypeError** –If data type of *reduction* is not string.
- **ValueError** –If *weight* or *pos_weight* can not be broadcast to a tensor with shape of *input*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> input = ms.Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]).astype(np.float32))
>>> target = ms.Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]).astype(np.float32))
>>> loss = mint.nn.BCEWithLogitsLoss()
>>> output = loss(input, target)
>>> print(output)
0.3463612
```

mindspore.mint.nn.CrossEntropyLoss

class mindspore.mint.nn.**CrossEntropyLoss**(*weight=None*, *ignore_index=- 100*, *reduction='mean'*, *label_smoothing=0.0*)

The cross entropy loss between input and target.

The cross entropy supports two kind of targets:

- Class indices (int) in the range $[0, C)$ where C is the number of classes, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}$$

where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

- Probabilities (float) for each class, useful when labels beyond a single class per minibatch item are required, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Note: Dynamic shape, dynamic rank and variable constant input are not supported in strict graph mode (jit_syntax_level=mindspore.STRICT).

Parameters

- **weight** (*Tensor, optional*) –A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products). Default: `None` .
- **ignore_index** (*int, optional*) –Specifies a target value that is ignored and does not contribute to the input gradient. Only valid in class indices, please set it to a negative number in probabilities. Default: `-100` .
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **label_smoothing** (*float, optional*) –Label smoothing values, a regularization tool used to prevent the model from overfitting when calculating Loss. The value range is [0.0, 1.0]. Default: `0.0` .

Inputs:

- **input** (Tensor) - (N) or (N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$. *input* is expected to be log-probabilities, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products).
- **target** (Tensor) - For class indices, tensor of shape (), (N) or $(N, d_1, d_2, ..., d_K)$, data type must be int32 or int64. For probabilities, tensor of shape $(N,)$, (N, C) or $(N, C, d_1, d_2, ..., d_K)$, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products).

Outputs:

Tensor, the data type is the same as *input* .

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> # Case 1: Indices labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.array([1, 0, 4]), ms.int32)
>>> op = ms.mint.nn.CrossEntropyLoss()
>>> output = op(inputs, target)
>>> # Case 2: Probability labels
>>> inputs = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> target = ms.Tensor(np.random.randn(3, 5), ms.float32)
>>> op = ms.mint.nn.CrossEntropyLoss()
>>> output = op(inputs, target)
```

mindspore.mint.nn.KLDivLoss

class mindspore.mint.nn.**KLDivLoss**(*reduction='mean'*, *log_target=False*)

Computes the Kullback-Leibler divergence between the *input* and the *target*.

For tensors of the same shape x and y, the updating formulas of KLDivLoss algorithm are as follows,

$$L(x, y) = y \cdot (\log y - x)$$

Then,

$$\ell(x, y) = \begin{cases} L(x, y), & \text{if reduction = 'none';} \\ \text{mean}(L(x, y)), & \text{if reduction = 'mean';} \\ \text{sum}(L(x, y))/x.\text{shape}[0], & \text{if reduction = 'batchmean';} \\ \text{sum}(L(x, y)), & \text{if reduction = 'sum'.} \end{cases}$$

where x represents *input*, y represents *target*, and $\ell(x, y)$ represents the output.

Note: The output aligns with the mathematical definition of Kullback-Leibler divergence only when *reduction* is set to `'batchmean'`.

Parameters

- **reduction** (*str, optional*) –Specifies the reduction to be applied to the output. Default: `'mean'`.
- **log_target** (*bool, optional*) –Specifies whether *target* is passed in the log space. Default: `False`.

Inputs:

- **input** (Tensor) - The input Tensor. The data type must be float16, float32 or bfloat16(only supported by Atlas A2 training series products).

- **target** (Tensor) - The target Tensor which has the same type as *input*. The shapes of *target* and *input* should be broadcastable.

Outputs:

Tensor, has the same dtype as *input*. If *reduction* is `'none'`, then output has the shape as broadcast result of the *input* and *target*. Otherwise, it is a scalar Tensor.

Raises

- **TypeError** – If neither *input* nor *target* is a Tensor.
- **TypeError** – If dtype of *input* or *target* is not float16, float32 or bfloat16.
- **TypeError** – If dtype of *target* is not the same as *input*.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'`, `'sum'`, `'batchmean'`.
- **ValueError** – If shapes of *target* and *input* can not be broadcastable.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> input = ms.Tensor(np.array([[0.5, 0.5], [0.4, 0.6]]), ms.float32)
>>> target = ms.Tensor(np.array([[0., 1.], [1., 0.]]), ms.float32)
>>> loss = mint.nn.KLDivLoss(reduction='mean', log_target=False)
>>> output = loss(input, target)
>>> print(output)
-0.225
```

mindspore.mint.nn.L1Loss

class mindspore.mint.nn.**L1Loss**(*reduction='mean'*)

L1Loss is used to calculate the mean absolute error between the predicted value and the target value.

Assuming that the x and y are 1-D Tensor, length N, then calculate the loss of x and y without dimensionality reduction (the reduction parameter is set to `'none'`). The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with } l_n = |x_n - y_n|,$$

where N is the batch size. If *reduction* is not `'none'`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.

- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Predicted value, Tensor of any dimension.
- **labels** (Tensor) - Target value, same shape as the *logits* in common cases. However, it supports the shape of *logits* is different from the shape of *labels* and they should be broadcasted to each other.

Outputs:

Tensor, data type is float.

Raises

- **ValueError** – If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.
- **ValueError** – If *logits* and *labels* have different shapes and cannot be broadcasted to each other.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = mint.nn.L1Loss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
0.33333334
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = mint.nn.L1Loss(reduction='none')
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0. 1. 2.]
 [0. 0. 1.]]
```

mindspore.mint.nn.MSELoss

class mindspore.mint.nn.**MSELoss**(*reduction='mean'*)

Calculates the mean squared error between the predicted value and the label value.

For simplicity, let x and y be 1-dimensional Tensor with length N, the unreduced loss (i.e. with argument reduction set to 'none') of x and y is given as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with} \quad l_n = (x_n - y_n)^2.$$

where N is the batch size. If *reduction* is not `'none'`, then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

reduction (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - The predicted value of the input. Tensor of any dimension. The data type needs to be consistent with the *labels*. It should also be broadcastable with the *labels*.
- **labels** (Tensor) - The input label. Tensor of any dimension. The data type needs to be consistent with the *logits*. It should also be broadcastable with the *logits*.

Outputs:

- Tensor. If *reduction* is `'mean'` or `'sum'`, the shape of output is *Tensor Scalar*.
- If reduction is `'none'`, the shape of output is the broadcasted shape of *logits* and *labels* .

Raises

- **ValueError** –If *reduction* is not one of `'mean'`, `'sum'` or `'none'`.
- **ValueError** –If *logits* and *labels* are not broadcastable.
- **TypeError** –If *logits* and *labels* are in different data type.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> # Case 1: logits.shape = labels.shape = (3,)
>>> loss = mint.nn.MSELoss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 1, 1]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
1.6666667
>>> # Case 2: logits.shape = (3,), labels.shape = (2, 3)
>>> loss = mint.nn.MSELoss(reduction='none')
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[[0. 1. 4.]
 [0. 0. 1.]]
```

mindspore.mint.nn.NLLLoss

class mindspore.mint.nn.**NLLLoss**(*weight=None*, *ignore_index=- 100*, *reduction='mean'*)

Gets the negative log likelihood loss between inputs and target.

The nll loss with reduction=none can be described as:

$$\ell(x,t) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_{t_n} x_{n,t_n}, \quad w_c = \text{weight}\,[c] \cdot \mathbb{1}\{c \neq \text{ignore\_index}\},$$

where x is the inputs, t is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `'None'` (default `'mean'`), then

$$\ell(x,t) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{tn}} l_n, & \text{if reduction} = \text{'mean'}, \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'} \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **weight** (*Tensor, optional*) – A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products). It should have the same data type as *input* . Default: `None` .
- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. Only valid in class indices, please set it to a negative number in probabilities. Default: `-100` .
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **input** (Tensor) - (N) or (N, C) where C = *number of classes* , N = *batch size* , or $(N, C, d_1, d_2, ..., d_K)$ (for high-dimensional data). *input* is expected to be log-probabilities, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products).
- **target** (Tensor) - () or (N) , where the value range is $[0, C-1]$, or $(N, d_1, d_2, ..., d_K)$ for high-dimensional loss, data type must be int32 or int64 or uint8.

Outputs:

Tensor, the data type is the same as *input* .

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> inputs = mindspore.Tensor(np.random.randn(3, 5), mindspore.float32)
>>> target = mindspore.Tensor(np.array([1, 0, 4]), mindspore.int32)
>>> op = mindspore.mint.nn.NLLLoss()
>>> output = op(inputs, target)
```

mindspore.mint.nn.SmoothL1Loss

class mindspore.mint.nn.**SmoothL1Loss**(*reduction='mean'*, *beta=1.0*)

Computes smooth L1 loss, a robust L1 loss.

Refer to *mindspore.mint.nn.functional.smooth_l1_loss()* for more details.

> **Warning:** This is an experimental API that is subject to change or deletion.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([2, 2, 3]), mindspore.float32)
>>> target = Tensor(np.array([2, 2, 2]), mindspore.float32)
>>> beta = 1.0
>>> reduction_1 = 'none'
>>> loss1 = mint.nn.SmoothL1Loss(reduction=reduction_1, beta=beta)
>>> output = loss1(input, target)
>>> print(output)
[0.  0.  0.5]
>>> reduction_2 = 'mean'
>>> loss2 = mint.nn.SmoothL1Loss(reduction=reduction_2, beta=beta)
>>> output = loss2(input, target)
>>> print(output)
0.16666667
>>> reduction_3 = 'sum'
>>> loss3 = mint.nn.SmoothL1Loss(reduction=reduction_3, beta=beta)
>>> output = loss3(input, target)
>>> print(output)
0.5
```

4.4.10 Vision Layer

API Name	Description	Supported Platforms
mindspore.mint.nn.PixelShuffle	Rearrange elements in a tensor according to an upscaling factor.	Ascend
mindspore.mint.nn.Upsample	For details, please refer to *mindspore.mint.nn.functional.interpolate()*.	Ascend

mindspore.mint.nn.PixelShuffle

class mindspore.mint.nn.**PixelShuffle**(*upscale_factor*)

Rearrange elements in a tensor according to an upscaling factor.

For details, please refer to *mindspore.mint.nn.functional.pixel_shuffle()*.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint
>>> pixel_shuffle = mint.nn.PixelShuffle(3)
>>> input = mint.randn(1, 9, 4, 4)
>>> output = pixel_shuffle(input)
>>> print(output.shape())
    [1, 1, 12, 12]
```

mindspore.mint.nn.Upsample

class mindspore.mint.nn.**Upsample**(*size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*)

For details, please refer to *mindspore.mint.nn.functional.interpolate()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> x = ms.Tensor([[[[1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0]]]])
>>> upsample = mint.nn.Upsample(size=(5, 5))
>>> out = upsample(x)
>>> print(x.asnumpy())
[[[[1. 2. 3. 4.]
```

(continues on next page)

(continued from previous page)

```
   [5. 6. 7. 8.]]]]
>>> print(out.asnumpy())
[[[[1. 1. 2. 3. 4.]
   [1. 1. 2. 3. 4.]
   [1. 1. 2. 3. 4.]
   [5. 5. 6. 7. 8.]
   [5. 5. 6. 7. 8.]]]]
>>> print(out.shape)
(1, 1, 5, 5)
```

4.4.11 Tools

API Name	Description	Supported Platforms
mindspore.mint.nn.Identity	A placeholder identity operator that returns the same as input.	Ascend

mindspore.mint.nn.Identity

class mindspore.mint.nn.**Identity**(**args*, ***kwargs*)

A placeholder identity operator that returns the same as input.

Parameters

- **args** (*Any*) – Any argument.
- **kwargs** (*Any*) – Any keyword argument.

Inputs:

- **input** (Any) - The input of Identity.

Outputs:

The same as *input*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([1, 2, 3, 4]), mindspore.int64)
>>> net = mint.nn.Identity()
>>> output = net(input)
>>> print(output)
[1 2 3 4]
```

4.5 mindspore.mint.nn.functional

4.5.1 Convolution functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.conv2d	Applies a 2D convolution over an input tensor.	Ascend
mindspore.mint.nn.functional.conv3d	Applies a 3D convolution over an input tensor.	Ascend
mindspore.mint.nn.functional.conv_transpose2d	Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called deconvolution (although it is not an actual deconvolution).	Ascend
mindspore.mint.nn.functional.fold	Combines an array of sliding local blocks into a large containing tensor.	Ascend
mindspore.mint.nn.functional.unfold	Extracts sliding local blocks from a batched input tensor.	Ascend

mindspore.mint.nn.functional.conv2d

mindspore.mint.nn.functional.**conv2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 2D convolution over an input tensor. The input tensor is typically of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}), where N is batch size, C is channel number, H is feature height, W is feature width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, , $weight$ is the convolution kernel value and X represents the input feature map.

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{out_j})$ represents the bias of the j-th output channel, $weight(C_{out_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1]), where kernel_size[0] and kernel_size[1] are the height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition and ConvNets.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in}).
- **weight** (Tensor) –Tensor of shape $(N, C_{in}/\text{groups}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, then the size of kernel is (kernel_size[0], kernel_size[1]).
- **bias** (Tensor, *optional*) –Bias Tensor with shape (C_{out}). When bias is None , zeros will be used. Default: None .
- **stride** (*Union(int, tuple[int], list[int])*, *optional*) –The distance of kernel moving, an int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: 1 .
- **padding** (*Union[int, tuple[int], list[int], str]*, *optional*) –The number of padding on the height and width directions of the input. The data type is an integer or a tuple of two integers or string {*valid*, *same*}. If *padding* is an integer, then *padding_{H}* and *padding_{W}* are all equal to *padding*. If *padding* is a tuple of 2 integers, then *padding_{H}* and *padding_{W}* is equal to *padding[0]* and *padding[1]* respectively. The value should be greater than or equal to 0. Default: 0 .
  - "same": Pad the input around its edges so that the shape of input and output are the same when *stride* is set to 1. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *stride* must be 1.
  - "valid": No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded.
- **dilation** (*Union(int, tuple[int], list[int])*, *optional*) –Gaps between kernel elements.The data type is int or a tuple of 2 integers. Specifies the dilation rate to use for dilated convolution. If set to be $k > 1$, there will be $k - 1$ pixels skipped for each sampling location. Its value must be greater than or equal to 1 and bounded by the height and width of the input *x*. Default: 1 .
- **groups** (*int*, *optional*) –Splits *input* into groups. Default: 1 .
  - (C_{in}

Returns

Tensor, the value that applied 2D convolution. The shape is $(N, C_{out}, H_{out}, W_{out})$. To see how different pad modes affect the output shape, please refer to *mindspore.mint.nn.Conv2d* for more details.

Raises

- **ValueError** –Args and size of the input feature map should satisfy the output formula to ensure that the size of the output feature map is positive; otherwise, an error will be reported. For more details on the output formula, please refer to *mindspore.mint.nn.Conv2d*.
- **RuntimeError** –On Ascend, due to the limitation of the L1 cache size of different NPU chip, if input size or kernel size is too large, it may trigger an error.
- **TypeError** –If *in_channels* , *out_channels* or *groups* is not an int.
- **TypeError** –If *kernel_size* , *stride* or *dilation* is neither an int nor a tuple.
- **TypeError** –If *bias* is not a Tensor.
- **ValueError** –If the shape of *bias* is not (C_{out}) .
- **ValueError** –If *stride* or *dilation* is less than 1.
- **ValueError** –If *padding* is *same* , *stride* is not equal to 1.
- **ValueError** –The input parameters do not satisfy the convolution output formula.
- **ValueError** –The KernelSize cannot exceed the size of the input feature map.

- **ValueError** –The value of padding cannot cause the calculation area to exceed the input size.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint, mint
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> output = mint.nn.functional.conv2d(x, weight)
>>> print(output.shape)
(10, 32, 30, 30)
```

mindspore.mint.nn.functional.conv3d

mindspore.mint.nn.functional.**conv3d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 3D convolution over an input tensor. The input tensor is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$ or $(C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, D, H, W are the depth, height and width of the feature graph, respectively.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation , $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kd, kh, kw) where kd , kd andkw are the depth, height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, kd, kh, kw)$, where *group* is the number of groups dividing x's input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

The following lists some of the limitations of the parameters.

- input –The input to the conv3d. The input must have each dimension size within the range [1, int32_max].
- weight –Filters of shape $(C_{out}, C_{in}/groups, kd, kh, kw)$. The value of kh and kw is in the range [1, 511]. The remaining values are in the range [1, int32_max]. And $kh * kw * k0$ is less 65536 (k0 is 16. If data type is float32, k0 is 8).

- bias −Bias Tensor with shape (C_{out}). The shape must equal the first dimension of the weight.
- stride −The distance of kernel moving. It can be an int number or tuple (noted by $(stride_d, stride_h, stride_w)$). stride_h and stride_w are in the range [1, 63]. stride_d is in the range [1, 255].
- padding −If padding is an int number, it is in the range [0, 255].
- dilation −The value is in the range [1, 255].
- groups −The value is in the range [1, 65535].
- C_{in}%groups == 0 and C_{out}%groups == 0 .
- $weight[1] == C_{in}/groups$.
- $H_{in} + PadUp + PadDown >= (kh - 1) * DilationH + 1$.
- $W_{in} + PadLeft + PadRight >= (kw - 1) * DilationW + 1$.
- $D_{in} + PadFront + PadBack >= (kd - 1) * DilationD + 1$.
- $H_{out} = (H_{in} + PadUp + PadDown - ((kh - 1) * DilationH + 1))/StrideH + 1$.
- $W_{out} = (W_{in} + PadLeft + PadRight - ((kw - 1) * DilationW + 1))/StrideW + 1$.
- $D_{out} = (D_{in} + PadFront + PadBack - ((kd - 1) * DilationD + 1))/StrideD + 1$.
- $(D_{in} + PadFront + PadBack - ((kd - 1) * DilationD + 1))/$
- $(H_{in} + PadUp + PadDown - ((kh - 1) * Dilationh + 1))/$
- $stride_d <= kernel_d$.
- $PadUp < kh$ and $PadDown < kh$. When *padding* = `'valid'`, both PadUp and PadDown are zeros. When *padding* = `'same'`, pad can be calculated by $floor(((H_{out} - 1) * strideH + (kh - 1) * DilationH + 1 - H_{in})/2)$ for high dimension. It is similar way to calculate the padding for depth and width dimension. And the depth and width dimensions also have the same constraints.
- $((kh - 1) * DilationH - PadUp)$ should be in [0, 255]. It is the same constraint for depth and width dimension.
- If *padding* is `'same'`, *stride* must be 1.

> **Warning:** This API does not support Atlas series products. This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$.
- **weight** (Tensor) −Set size of kernel is (kd, kh, kw), then the shape is $(C_{out}, C_{in}/groups, kd, kh, kw)$.
- **bias** (Tensor, *optional*) −Bias Tensor with shape (C_{out}). When bias is `None` , zeros will be used. Default: `None` .
- **stride** (*Union(int, tuple[int], list[int]), optional*) −The distance of kernel moving, an int number that represents the depth, the height and width of movement are both strides, or a tuple of triple int numbers that represent the depth, height and width of movement respectively. Default: `1` .
- **padding** (*Union(int, tuple[int], list[int], str), optional*) −Implicit paddings on both sides of the input *x*. Can be a string, one integer or a tuple/list with 3 integers. If *padding* is a string, the optional values are `"same"` , `"valid"`.
  - same: Adopts the way of completion. The height and width of the output will be equal to the input *x* divided by stride. The padding will be evenly calculated in top and bottom, left and right possiblily. Otherwise, the last extra padding will be calculated from the bottom and the right side. If this mode is set, *stride* must be 1.

– valid: Adopts the way of discarding. The possible largest height and width of output will be returned without padding. Extra pixels will be discarded.

If *padding* is one integer, the paddings of top, bottom, left and right are the same, equal to padding. If *padding* is a tuple/list with 3 integers, the padding of head, tail, top, bottom, left and right equal to pad[0], pad[0], pad[1], pad[1], pad[2] and pad[2] correspondingly. Default: `0` .

- **dilation** (*Union[int, tuple[int], list[int]], optional*) – Controlling the space between the kernel points. Default: `1` .
- **groups** (*int, optional*) – Splits *input* into groups. Default: `1` .

Returns

Tensor, the same dtype as the *input*, with the shape $(N, C_{out}, D_{out}, H_{out}, W_{out})$ or $(C_{out}, D_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** – If *stride*, *padding* or *dilation* is neither an int nor a tuple.
- **TypeError** – *groups* is not an int.
- **TypeError** – If *bias* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import mint
>>> x = mindspore.Tensor(np.random.randn(12, 1, 60, 50, 8), mindspore.float16)
>>> w = mindspore.Tensor(np.random.randn(26, 1, 2, 4, 4), mindspore.float16)
>>> out = mint.nn.functional.conv3d(x, w)
>>> print(out.shape)
(12, 26, 59, 47, 5)
```

mindspore.mint.nn.functional.conv_transpose2d

mindspore.mint.nn.functional.**conv_transpose2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *dilation=1*)

Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called deconvolution (although it is not an actual deconvolution).

Refer to *mindspore.mint.nn.ConvTranspose2d* for more details.

Warning:

- This is an experimental API that is subject to change or deletion.
- In the scenario where inputs are non-contiguous, *output_padding* must be less than *stride* .
- For Atlas training products, when the dtype of input is float32, the *groups* only supports 1.

Parameters

- **input** (Tensor) –input tensor of shape $(minibatch, in\_channels, iH, iW)$ or $(in\_channels, iH, iW)$.
- **weight** (Tensor) –filters of shape $(in\_channels, \frac{out\_channels}{\text{groups}}, kH, kW)$.
- **bias** (Tensor, *optional*) –bias of shape $(out\_channels)$. Default: None .
- **stride** (*Union[int, tuple(int), list[int]], optional*) –the stride of the convolving kernel. Can be a single number or a tuple (sH, sW) . Default: 1 .
- **padding** (*Union[int, tuple(int), list[int]], optional*) –$dilation * (kernel\_size - 1) - padding$ zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple $(padH, padW)$. Default: 0 .
- **output_padding** (*Union[int, tuple(int), list[int]], optional*) –additional size added to one side of each dimension in the output shape. Can be a single number or a tuple (out_padH, out_padW) . The value of *output_padding* must be less than *stride* or *dilation* . Default: 0 .
- **groups** (*int, optional*) –split input into groups, $in\_channels$ should be divisible by the number of groups. Default: 1 .
- **dilation** (*Union[int, tuple(int), list[int]], optional*) –the spacing between kernel elements. Can be a single number or a tuple (dH, dW) . Default: 1 .

Returns

Tensor of shape $(minibatch, out\_channels, oH, oW)$ or $(out\_channels, oH, oW)$, where

$$oH = (iH - 1) \times sH - 2 \times padH + dH \times (kH - 1) + out\_padH + 1$$

$$oW = (iW - 1) \times sW - 2 \times padW + dW \times (kW - 1) + out\_padW + 1$$

Raises

- **TypeError** –If *stride*, *padding*, *output_padding* or *dilation* is neither an int nor a tuple or a list.
- **TypeError** –If *groups* is not an int.
- **ValueError** –If the shape of *bias* is not $(out\_channels)$.
- **ValueError** –If *stride* or *dilation* is less than 1.
- **ValueError** –If *padding* or *output_padding* is less than 0.
- **ValueError** –If *stride*, *padding*, *output_padding* or *dilation* is a tuple whose length is not equal to 2.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.ones([1, 4, 5, 5]), mindspore.float32)
>>> weight = Tensor(np.ones([4, 8, 3, 3]), mindspore.float32)
>>> output = mint.nn.functional.conv_transpose2d(x, weight)
>>> print(output.shape)
(1, 8, 7, 7)
```

mindspore.mint.nn.functional.fold

mindspore.mint.nn.functional.**fold**(*input*, *output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Combines an array of sliding local blocks into a large containing tensor.

Consider a batched input tensor of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where N is the batch dimension, $C \times \prod(\text{kernel\_size})$ is the total number of values within each block (a block has $\prod(\text{kernel\_size})$ spatial locations each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_{d} \left\lfloor \frac{\text{output\_size}[d] + 2 \times \text{padding}[d] - \text{dilation}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{stride}[d]} + 1 \right\rfloor,$$

where d is over all spatial dimensions.

Therefore, *output_size* is the spatial shape of the large containing tensor of the sliding local blocks.

The *dilation*, *padding* and *stride* arguments specify how the sliding blocks are retrieved.

> **Warning:** Currently, only unbatched(3D) or batched(4D) image-like output tensors are supported.

Parameters

- **input** (Tensor) −2-D or 3-D Tensor.
- **output_size** (*Union[int, tuple[int], list[int]]*) −The shape of the spatial dimensions of the output(i.e., output.shape[2:]).
- **kernel_size** (*Union[int, tuple[int], list[int]]*) −The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width. Must be specified.
- **dilation** (*Union[int, tuple[int], list[int]], optional*) −The size of the dilation, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **padding** (*Union[int, tuple[int], list[int]], optional*) −The size of the padding, should be two int for height and width. If type is int, it means that height equal with width. Default: `0` .
- **stride** (*Union[int, tuple[int], list[int]], optional*) −The size of the stride, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .

Returns

A Tensor, with same type as *input* .

Shape:

- Input: $(N, C \times \prod(\text{kernel\_size}), L)$ or $(C \times \prod(\text{kernel\_size}), L)$
- Output: $(N, C, output\_size[0], output\_size[1], ...)$ or $(C, output\_size[0], output\_size[1], ...)$

Raises

- **TypeError** −If *output_size*, *kernel_size*, *stride*, *dilation*, *padding* data type is not int, tuple or list.
- **ValueError** −If *output_size*, *kernel_size*, *dilation*, *stride* value is not greater than zero or elements number invalid.
- **ValueError** −If *padding* value is less than zero or elements number invalid.
- **ValueError** −If input.shape[-2] can't be divisible by the product of kernel_size.
- **ValueError** −If *input.shape[-1]* is not equal to the calculated number of sliding blocks *L*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.random.rand(16, 64, 25).astype(np.float32))
>>> output = mint.nn.functional.fold(x, (8, 8), [2, 2], [2, 2], [2, 2], [2, 2])
>>> print(output.shape)
(16, 16, 8, 8)
```

mindspore.mint.nn.functional.unfold

mindspore.mint.nn.functional.**unfold**(*input*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Extracts sliding local blocks from a batched input tensor.

Consider a batched input tensor of shape $(N, C, *)$, where N is the batch dimension, C is the channel dimension, and $*$ represent arbitrary spatial dimensions. This operation flattens each sliding *Kernel_size*- sized block within the spatial dimensions of *input* into a column (i.e., last dimension) of a 3-D output tensor of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where $C \times \prod(\text{kernel\_size})$ is the total number of values within each block (a block has $\prod(\text{kernel\_size})$ spatial locations each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_{d} \left\lfloor \frac{\text{spatial\_size}[d] + 2 \times \text{padding}[d] - \text{dilation}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{stride}[d]} + 1 \right\rfloor,$$

where spatial_size is formed by the spatial dimensions of *input* ($*$ above), and d is over all spatial dimensions.

Therefore, indexing *output* at the last dimension (column dimension) gives all values within a certain block.

The *dilation*, *padding* and *stride* arguments specify how the sliding blocks are retrieved.

Warning:

- Currently, batched(4D) image-like tensors are supported.
- For Ascend, it is only supported on platforms above Atlas A2.

Parameters

- **input** (Tensor) – 4-D Tensor.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) – The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width. Must be specified.
- **dilation** (*Union[int, tuple[int], list[int]], optional*) – The dilation of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **padding** (*Union[int, tuple[int], list[int]], optional*) – The pad of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: `0` .
- **stride** (*Union[int, tuple[int], list[int]], optional*) – The stride of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .

Returns

A Tensor, with same type as *input* .

Shape:

- Input: $(N, C, *)$
- Output: $(N, C \times \prod(\text{kernel\_size}), L)$

Raises

- **TypeError** – If any data type of *kernel_size*, *stride*, *dilation*, *padding* is not int, tuple or list.
- **ValueError** – If *kernel_size*, *dilation*, *stride* value is not greater than zero or elements number more than *2*.
- **ValueError** – If *padding* value is less than zero.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.random.rand(4, 4, 32, 32), mindspore.float32)
>>> output = mint.nn.functional.unfold(x, kernel_size=3, dilation=1, stride=1)
>>> print(output.shape)
(4, 36, 900)
```

4.5.2 Pooling functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.adaptive_avg_pool1d	Performs 1D adaptive average pooling on a multi-plane input signal.	`Ascend`
mindspore.mint.nn.functional.adaptive_avg_pool2d	Performs 2D adaptive average pooling on a multi-plane input signal.	`Ascend`
mindspore.mint.nn.functional.adaptive_avg_pool3d	Performs 3D adaptive average pooling on a multi-plane input signal.	`Ascend`
mindspore.mint.nn.functional.adaptive_max_pool1d	Performs 1D adaptive max pooling on a multi-plane input signal.	`Ascend`
mindspore.mint.nn.functional.avg_pool1d	Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.	`Ascend`
mindspore.mint.nn.functional.avg_pool2d	Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes.	`Ascend`
mindspore.mint.nn.functional.avg_pool3d	Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes.	`Ascend`
mindspore.mint.nn.functional.max_pool2d	Performs a 2D max pooling on the input Tensor.	`Ascend`
mindspore.mint.nn.functional.max_unpool2d	Computes the inverse of *max_pool2d*.	`Ascend`

mindspore.mint.nn.functional.adaptive_avg_pool1d

mindspore.mint.nn.functional.**adaptive_avg_pool1d**(*input*, *output_size*)

Performs 1D adaptive average pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is L. The number of output features is equal to the number of input features.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input of adaptive_avg_pool1d, which is a 2D or 3D tensor, with float16 or float32 data type.
- **output_size** (*int*) –The target output feature size. *output_size* is an integer.

Returns

Tensor, with the same type as the *input*.

Shape of the output is *input_shape[:len(input_shape) - 1] + [output_size]*.

Raises

- **ValueError** –If *output_size* is not integer.
- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[2,3],[3,4]],dtype=mindspore.float16)
>>> output = mint.nn.functional.adaptive_avg_pool1d(input, 3)
>>> print(output)
[[2.  2.5 3. ]
 [3.  3.5 4. ]]
```

mindspore.mint.nn.functional.adaptive_avg_pool2d

mindspore.mint.nn.functional.**adaptive_avg_pool2d**(*input*, *output_size*)

Performs 2D adaptive average pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is H x W. The number of output features is equal to the number of input features.

The input and output data format can be "NCHW" and "CHW". N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

For adaptive average pooling for 2D:

$$
\begin{aligned}
h_{start} &= floor(i * H_{in}/H_{out}) \\
h_{end} &= ceil((i+1) * H_{in}/H_{out}) \\
w_{start} &= floor(j * W_{in}/W_{out}) \\
w_{end} &= ceil((j+1) * W_{in}/W_{out}) \\
Output(i,j) &= \frac{\sum Input[h_{start}:h_{end}, w_{start}:w_{end}]}{(h_{end}-h_{start})*(w_{end}-w_{start})}
\end{aligned}
$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input of adaptive_avg_pool2d, which is a 3D or 4D tensor, with float16 or float32 data type.
- **output_size** (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (H, W), or an int H for (H, H). H and W can be int or None. If it is None, it means the output size is the same as the input size.

Returns

Tensor, with the same type as the *input*.

Shape of the output is *input_shape[:len(input_shape) - len(out_shape)] + out_shape*.

$$
out\_shape = \begin{cases}
input\_shape[-2] + output\_size[1], & \text{if } output\_size \text{ is (None, w);} \\
output\_size[0] + input\_shape[-1], & \text{if } output\_size \text{ is (h, None);} \\
input\_shape[-2:], & \text{if } output\_size \text{ is (None, None);} \\
(h, h), & \text{if } output\_size \text{ is h;} \\
(h, w), & \text{if } output\_size \text{ is (h, w)}
\end{cases}
$$

Raises

- **ValueError** –If *output_size* is a tuple and the length of *output_size* is not 2.
- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32 or float64.
- **ValueError** –If the dimension of *input* is less than or equal to the dimension of *output_size*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> # case 1: output_size=(3, 2)
>>> input = Tensor(np.array([[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                          [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                          [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]),␣
↪mindspore.float32)
>>> output = mint.nn.functional.adaptive_avg_pool2d(input, (3, 2))
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]]
```

mindspore.mint.nn.functional.adaptive_avg_pool3d

mindspore.mint.nn.functional.**adaptive_avg_pool3d**(*input*, *output_size*)

Performs 3D adaptive average pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is (D, H, W). The number of output features is equal to the number of input planes.

Suppose the last 3 dimension size of x is (D_{in}, H_{in}, W_{in}), the last 3 dimension size of output is $(D_{out}, H_{out}, W_{out})$.

$$
\begin{aligned}
&\forall \quad od \in [0, D_{out} - 1], oh \in [0, H_{out} - 1], ow \in [0, W_{out} - 1] \\
&output[od, oh, ow] = \\
&\qquad mean(x[D_{istart} : D_{iend} + 1, H_{istart} : H_{iend} + 1, W_{istart} : W_{iend} + 1]) \\
&where, \\
&\qquad D_{istart} = \left\lceil \frac{od * D_{in}}{D_{out}} \right\rceil \\
&\qquad D_{iend} = \left\lfloor \frac{(od+1) * D_{in}}{D_{out}} \right\rfloor \\
&\qquad H_{istart} = \left\lceil \frac{oh * H_{in}}{H_{out}} \right\rceil \\
&\qquad H_{iend} = \left\lfloor \frac{(oh+1) * H_{in}}{H_{out}} \right\rfloor \\
&\qquad W_{istart} = \left\lceil \frac{ow * W_{in}}{W_{out}} \right\rceil \\
&\qquad W_{iend} = \left\lfloor \frac{(ow+1) * W_{in}}{W_{out}} \right\rfloor
\end{aligned}
$$

Warning: For Ascend, it is only supported on Atlas A2 Training Series Products. This is an experimental optimizer API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input of adaptive_avg_pool3d, which is a 4D or 5D Tensor.
- **output_size** (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (D, H, W), or an int D for (D, D, D). D, H and W can be int or None which means the output size is the same as that of the input.

Returns

Tensor, with the same type as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **ValueError** –If the dimension of *input* is not 4D or 5D.
- **ValueError** –If *output_size* value is not positive.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> # case 1: output_size=(3, 3, 4)
>>> output_size=(3, 3, 4)
>>> input_val = np.random.randn(4, 3, 5, 6, 7)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = mint.nn.functional.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(4, 3, 3, 3, 4)
>>> # case 2: output_size=4
>>> output_size=5
>>> input_val = np.random.randn(2, 3, 8, 6, 12)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = mint.nn.functional.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(2, 3, 5, 5, 5)
>>> # case 3: output_size=(None, 4, 5)
>>> output_size=(None, 4, 5)
>>> input_val = np.random.randn(4, 1, 9, 10, 8)
>>> input = Tensor(input_val, mindspore.float32)
>>> output = mint.nn.functional.adaptive_avg_pool3d(input, output_size)
>>> print(output.shape)
(4, 1, 9, 4, 5)
```

mindspore.mint.nn.functional.adaptive_max_pool1d

mindspore.mint.nn.functional.**adaptive_max_pool1d**(*input*, *output_size*, *return_indices=False*)

Performs 1D adaptive max pooling on a multi-plane input signal. That is, for any input size, the size of the specified output is L. The number of output features is equal to the number of input features.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (`Tensor`) −The input of adaptive_max_pool1d, which is a 2D or 3D tensor, with float16, float32 or float64 data type.
- **output_size** (*int*) −The target output feature size. *output_size* is an integer.
- **return_indices** (*bool, optional*) −Whether to return the index of the maximum value. Default: `False`.

Returns

Union(Tensor, tuple(Tensor, Tensor)).

- If *return_indices* is False, output is a Tensor, with shape (N, C, L_{out}). It has the same data type as *input*.
- If *return_indices* is True, output is a Tuple of 2 Tensors, representing the result and where the max values are generated.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or float64.
- **TypeError** – If *output_size* is not int or tuple.
- **TypeError** – If *return_indices* is not a bool.
- **ValueError** – If *output_size* is a tuple and the length of *output_size* is not 1.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[2,3],[3,4]],dtype=mindspore.float16)
>>> output = mint.nn.functional.adaptive_max_pool1d(input, 3)
>>> print(output)
[[2.  3.  3. ]
 [3.  4.  4. ]]
```

mindspore.mint.nn.functional.avg_pool1d

mindspore.mint.nn.functional.**avg_pool1d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 1D average pooling over an input Tensor which can be regarded as a composition of 1D input planes.

Typically the input is of shape (N_{in}, C_{in}, L_{in}), avg_pool1d outputs regional average in the (L_{in})-dimension. Given kernel size as $ks = l_{ker}$ and *stride* as $s = s_0$, the operation is as follows.

$$\text{output}(N_i, C_j, l) = \frac{1}{l_{ker}} \sum_{n=0}^{l_{ker}-1} \text{input}(N_i, C_j, s_0 \times l + n)$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – Tensor of shape (N, C_{in}, L_{in}).
- **kernel_size** (*Union(int, tuple[int])*) – The size of kernel window used to take the average value.
- **stride** (*Union(int, tuple[int]), optional*) – The distance of kernel moving. *stride* can either be an int number or a tuple of one int number. Default: `None`, the same value as *kernel_size*.
- **padding** (*Union(int, tuple[int]), optional*) – The pad length to be filled. *padding* can either be an integer or a tuple of one integer. Default: `0` .
- **ceil_mode** (*bool, optional*) – If True, apply ceil instead of floor to compute the output shape. Default: `False`.
- **count_include_pad** (*bool, optional*) – If True, include the zero-padding in the averaging calculation. Default: `True` .

Returns

Tensor of shape (N, C_{in}, L_{out}).

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *kernel_size* or *stride* is not an int.
- **TypeError** –If *ceil_mode* or *count_include_pad* is not a bool.
- **ValueError** –If *kernel_size* or *stride* is less than *1*.
- **ValueError** –If *kernel_size* or *stride* or *padding* is not int nor a tuple whose length is greater than *1*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.random.randint(0, 10, [1, 3, 6]), mindspore.float32)
>>> output = mint.nn.functional.avg_pool1d(input_x, kernel_size=6, stride=1)
>>> print(output.shape)
(1, 3, 1)
```

mindspore.mint.nn.functional.avg_pool2d

mindspore.mint.nn.functional.**avg_pool2d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*)

Applies a 2D average pooling over an input Tensor which can be regarded as a composition of 2D input planes. Typically the input is of shape (N, C, H_{in}, W_{in}) , outputs regional average in the (H_{in}, W_{in}) -dimension. Given kernel size (kH, kW) and *stride* , the operation is as follows.

$$\text{output}(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \text{input}(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Note: On the Atlas platform, when calculating the input, the precision is degraded from float32 to float16.

Parameters

- **input** (`Tensor`) –Tensor of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- **kernel_size** (*Union[int, tuple[int], list[int]]*) –The size of kernel used to take the average value. Can be a single number or a tuple (kH, kW) .
- **stride** (*Union[int, tuple[int], list[int]], optional*) –The distance of kernel moving. Can be a single number or a tuple (sH, sW) . Default: `None`, where its value is equal to *kernel_size*.
- **padding** (*Union[int, tuple[int], list[int]], optional*) –Implicit zero padding to be added on both sides. Can be a single number or a tuple $(padH, padW)$. Default: `0`.
- **ceil_mode** (*bool, optional*) –If True, apply ceil instead of floor to compute the output shape. Default: `False`.

- **count_include_pad** (*bool, optional*) –If True, include the zero-padding in the averaging calculation. Default: `True` .
- **divisor_override** (*int, optional*) –If specified, it will be used as divisor in the averaging calculation, otherwise size of pooling region will be used. Default: `None`.

Returns

Tensor, with shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}).

$$H_{out} = \frac{H_{in}+2\times padding[0]-kernel\_size[0]}{stride[0]} + 1$$
$$W_{out} = \frac{W_{in}+2\times padding[1]-kernel\_size[1]}{stride[1]} + 1$$

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *kernel_size* or *stride* is neither int nor tuple.
- **TypeError** –If *ceil_mode* or *count_include_pad* is not a bool.
- **TypeError** –If *divisor_override* is not an int or None.
- **ValueError** –If the dimension of *input* is not equal to *3* or *4*.
- **ValueError** –If *kernel_size* or *stride* is less than 1.
- **ValueError** –If value of *padding* is less than *0*.
- **ValueError** –If *kernel_size*, *padding* or *stride* is a tuple whose length is not equal to *1* or *2*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.arange(1 * 3 * 3 * 4).reshape(1, 3, 3, 4), mindspore.float32)
>>> output = mint.nn.functional.avg_pool2d(x, kernel_size=2, stride=1)
>>> print(output)
[[[[ 2.5   3.5   4.5]
   [ 6.5   7.5   8.5]]
  [[14.5  15.5  16.5]
   [18.5  19.5  20.5]]
  [[26.5  27.5  28.5]
   [30.5  31.5  32.5]]]]
```

mindspore.mint.nn.functional.avg_pool3d

mindspore.mint.nn.functional.**avg_pool3d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=None*)

Applies a 3D average pooling over an input Tensor which can be regarded as a composition of 3D input planes. Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$, outputs regional average in the (D_{in}, H_{in}, W_{in}) -dimension. Given kernel size (kD, kH, kW) and *stride* , the operation is as follows.

$$\text{output}(N_i, C_j, d, h, w) = \frac{1}{kD * kH * kW} \sum_{l=0}^{kD-1} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1}$$
$$\text{input}(N_i, C_j, stride[0] \times d + l, stride[1] \times h + m, stride[2] \times w + n)$$

Warning: This is an experimental API that is subject to change or deletion.

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **input** (Tensor) –Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(C, D_{in}, H_{in}, W_{in})$.
- **kernel_size** (*Union[int, tuple[int], list[int]]*) –The size of kernel used to take the average value. Can be a single number or a tuple (kD, kH, kW) .
- **stride** (*Union[int, tuple[int], list[int]], optional*) –The distance of kernel moving. Can be a single number or a tuple (sD, sH, sW) . Default: None, where its value is equal to *kernel_size*.
- **padding** (*Union[int, tuple[int], list[int]], optional*) –Implicit zero padding to be added on both sides. Can be a single number or a tuple $(padD, padH, padW)$. Default: 0.
- **ceil_mode** (*bool, optional*) –If True, apply ceil instead of floor to compute the output shape. Default: False.
- **count_include_pad** (*bool, optional*) –If True, include the zero-padding in the averaging calculation. Default: True .
- **divisor_override** (*int, optional*) –If specified, it will be used as divisor in the averaging calculation, otherwise size of pooling region will be used. Default: None.

Returns

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(C, D_{out}, H_{out}, W_{out})$.

$$D_{out} = \frac{D_{in}+2\times padding[0]-kernel\_size[0]}{stride[0]} + 1$$
$$H_{out} = \frac{H_{in}+2\times padding[1]-kernel\_size[0]}{stride[1]} + 1$$
$$W_{out} = \frac{W_{in}+2\times padding[2]-kernel\_size[1]}{stride[2]} + 1$$

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *kernel_size* or *stride* is neither int nor tuple.
- **TypeError** –If *ceil_mode* or *count_include_pad* is not a bool.

- **TypeError** –If *divisor_override* is not an int or None.
- **ValueError** –If the dimension of *input* is not equal to *4* or *5*.
- **ValueError** –If *kernel_size* or *stride* is less than 1.
- **ValueError** –If value of *padding* is less than *0*.
- **ValueError** –If *kernel_size*, *padding* or *stride* is a tuple whose length is not equal to *1* or *3*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.arange(1 * 2 * 2 * 2 * 3).reshape((1, 2, 2, 2, 3)), mindspore.
↪float16)
>>> output = mint.nn.functional.avg_pool3d(input_x, kernel_size=2, stride=1)
>>> print(output)
[[[[[ 5.  6.]]]
  [[[17. 18.]]]]]
```

mindspore.mint.nn.functional.max_pool2d

mindspore.mint.nn.functional.**max_pool2d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *ceil_mode=False*, *return_indices=False*)

Performs a 2D max pooling on the input Tensor.

Typically, the input is a Tensor with shape $(N_{in}, C_{in}, H_{in}, W_{in})$, outputs regional maximum in the (H_{in}, W_{in})-dimension. Given *kernel_size* $ks = (h_{ker}, w_{ker})$ and *stride* $s = (s_0, s_1)$, the operation is as follows:

$$\text{output}(N_i, C_j, h, w) = \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times h + m, s_1 \times w + n)$$

Warning: Only support on Atlas A2 training series.

Parameters

- **input** (Tensor) –Tensor of shape $(N_{in}, C_{in}, H_{in}, W_{in})$ with data type of float32 in Ascend.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value and arg value, is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int], None], optional*) –The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: `None` , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]], optional*) –An int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: `0` .

- **dilation** (*Union[int, tuple[int]], optional*) – Control the stride of elements in the kernel. Default: `1` .
- **ceil_mode** (*bool, optional*) – Whether to use ceil instead of floor to calculate output shape. Default: `False` .
- **return_indices** (*bool, optional*) – Whether to output the indices of max value. Default: `False` .

Returns

If *return_indices* is `False` , return a Tensor *output*, else return a tuple (*output*, *argmax*).

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, H_{out}, W_{out})$. It has the same data type as *input*.

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

- **argmax** (Tensor) - Index corresponding to the maximum value. In Ascend, data type is int32. It will be return only when *return_indices* is True.

Raises

- **TypeError** – If *input* is not a Tensor.
- **ValueError** – If length of shape of *input* is not equal to 4.
- **TypeError** – If *kernel_size* , *stride* , *padding* or *dilation* is not int or tuple.
- **ValueError** – If *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *dilation* is not all 1.
- **ValueError** – If *padding* is less than 0.
- **ValueError** – If *padding* is more than half of *kernel_size*.
- **TypeError** – If *ceil_mode* is not bool.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.arange(20 * 16 * 50 * 32).reshape((20, 16, 50, 32)), mindspore.float32)
>>> output_tensor, argmax = mint.nn.functional.max_pool2d(input, kernel_size=(3, 2),␣
↪stride=(2, 1),
...                                                                    ceil_mode=False, return_
↪indices=True)
>>> print(output_tensor.shape)
(20, 16, 24, 31)
>>> print(argmax.shape)
(20, 16, 24, 31)
```

mindspore.mint.nn.functional.max_unpool2d

mindspore.mint.nn.functional.**max_unpool2d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

Computes the inverse of *max_pool2d*.

max_unpool2d keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}), and the output is of shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}). The operation is as follows.

$$H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel\_size[0]$$
$$W_{out} = (W_{in} - 1) \times stride[1] - 2 \times padding[1] + kernel\_size[1]$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –The input Tensor to invert. Tensor of shape (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- **indices** (Tensor) –Max values' index represented by the indices. Tensor of shape must be same with input 'input'. Values of indices must belong to $[0, H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value, an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **stride** (*Union[int, tuple[int]], optional*) –The distance of kernel moving, an int number that represents the height and width of movement are both stride, or a tuple of two int numbers that represent height and width of movement respectively. Default: None , which indicates the moving step is *kernel_size* .
- **padding** (*Union[int, tuple[int]], optional*) –The pad value to be filled. Default: 0 . If *padding* is an integer, the paddings of height and width are the same, equal to padding. If *padding* is a tuple of two integers, the padding of height and width equal to padding[0] and padding[1] correspondingly.
- **output_size** (*tuple[int], optional*) –The target output size. Default: None . If output_size == (), then the shape of output computed by *kernel_size*, *stride* and *padding*. If output_size != (), then output_size must be (N, C, H, W) , (C, H, W) or (H, W) and output_size must belong to $[(N, C, H_{out} - stride[0], W_{out} - stride[1]), (N, C, H_{out} + stride[0], W_{out} + stride[1])]$.

Returns

Tensor, with shape (N, C, H_{out}, W_{out}) or (C, H_{out}, W_{out}), with the same data type with *input*.

Raises

- **TypeError** –If data type of *input* or *indices* is not supported.
- **TypeError** –If *kernel_size*, *stride* or *padding* is neither an int nor a tuple.
- **ValueError** –If numbers in *stride*, *padding* or *kernel_size* are not positive.
- **ValueError** –If the shapes of *input* and *indices* are different.
- **ValueError** –If the length of *input* is not 3 or 4.
- **ValueError** –If the type of *output_size* is not tuple.
- **ValueError** –If *output_size* is not close to output size computed by attr *kernel_size*, *stride*, *padding*.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[[[0, 1], [8, 9]]]]).astype(np.float32))
>>> indices = Tensor(np.array([[[[0, 1], [2, 3]]]]).astype(np.int64))
>>> output = mint.nn.functional.max_unpool2d(input, indices, 1, stride=1, padding=0)
>>> print(output.asnumpy())
[[[[0. 1.]
   [8. 9.]]]]
```

4.5.3 Non-linear activation functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.batch_norm	Batch Normalization for input data and updated parameters.	Ascend
mindspore.mint.nn.functional.elu	Exponential Linear Unit activation function	Ascend
*mindspore.mint.nn.functional.elu_*	Exponential Linear Unit activation function	Ascend
mindspore.mint.nn.functional.gelu	Gaussian Error Linear Units activation function.	Ascend
mindspore.mint.nn.functional.glu	Computes GLU (Gated Linear Unit activation function) of the input tensor.	Ascend
mindspore.mint.nn.functional.group_norm	Group Normalization over a mini-batch of inputs.	Ascend
mindspore.mint.nn.functional.hardshrink	Hard Shrink activation function.	Ascend
mindspore.mint.nn.functional.hardsigmoid	Hard Sigmoid activation function.	Ascend
mindspore.mint.nn.functional.hardswish	Hard Swish activation function.	Ascend
mindspore.mint.nn.functional.layer_norm	Applies the Layer Normalization on the mini-batch input.	Ascend
mindspore.mint.nn.functional.leaky_relu	leaky_relu activation function.	Ascend
mindspore.mint.nn.functional.log_softmax	Applies the Log Softmax function to the input tensor on the specified axis.	Ascend
mindspore.mint.nn.functional.logsigmoid	Applies logsigmoid activation element-wise.	Ascend
mindspore.mint.nn.functional.mish	Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.	Ascend
mindspore.mint.nn.functional.prelu	Parametric Rectified Linear Unit activation function.	Ascend
mindspore.mint.nn.functional.relu	Computes ReLU (Rectified Linear Unit activation function) of input tensors element-wise.	Ascend
mindspore.mint.nn.functional.relu6	Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input tensors element-wise.	Ascend
*mindspore.mint.nn.functional.relu_*	ReLuComputes ReLU (Rectified Linear Unit activation function) inplace of input tensors element-wise.	Ascend

continues on next page

Table 22 – continued from previous page

mindspore.mint.nn.functional.selu	Activation function SELU (Scaled exponential Linear Unit).	Ascend
mindspore.mint.nn.functional.sigmoid	Computes Sigmoid of input element-wise.	Ascend
mindspore.mint.nn.functional.silu	Computes Sigmoid Linear Unit of input element-wise, also known as Swish function.	Ascend
mindspore.mint.nn.functional.softmax	Applies the Softmax operation to the input tensor on the specified axis.	Ascend
mindspore.mint.nn.functional.softplus	Applies softplus function to *input* element-wise.	Ascend
mindspore.mint.nn.functional.softshrink	Soft Shrink activation function.	Ascend
mindspore.mint.nn.functional.tanh	Computes hyperbolic tangent of input element-wise.	Ascend

mindspore.mint.nn.functional.batch_norm

mindspore.mint.nn.functional.**batch_norm**(*input*, *running_mean*, *running_var*, *weight=None*, *bias=None*, *training=False*, *momentum=0.1*, *eps=1e-5*)

Batch Normalization for input data and updated parameters.

Batch Normalization is widely used in convolutional neural networks. This operation applies Batch Normalization over inputs to avoid internal covariate shift as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the features using a mini-batch of data and the learned parameters can be described in the following formula,

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

where γ is *weight*, β is *bias*, ϵ is *eps*, $mean$ is the mean of x, $variance$ is the variance of x.

Parameters

- **input** (Tensor) –Tensor of shape $(N, C, *)$, with bfloat16, float16 or float32 data type. For Atlas training products, the shape must be 2-4 dimensions currently.
- **running_mean** (Tensor) –The shape $(C,)$, with bfloat, float16 or float32 data type.
- **running_var** (Tensor) –The shape $(C,)$, with bfloat, float16 or float32 data type.
- **weight** (Tensor, *optional*) –The shape $(C,)$, with bfloat, float16 or float32 data type, Default: None. Initialized to 1 when *weight* is None.
- **bias** (Tensor, *optional*) –The shape $(C,)$, with bfloat, float16 or float32 data type. Default: None. Initialized to 0 when *weight* is None.
- **training** (*bool*, *optional*) –If *training* is *True*, *mean* and *variance* are computed during training. If *training* is *False*, they're loaded from checkpoint during inference. Default: False .
- **momentum** (*float*, *optional*) –The hyper parameter to compute moving average for *running_mean* and *running_var* (e.g. $new\_running\_mean = (1 - momentum) * running\_mean + momentum * current\_mean$). Default: 0.1 .
- **eps** (*float*, *optional*) –A small value added for numerical stability. Default: 1e-5.

Returns

Tensor, has the same type and shape as *input*. The shape is $(N, C, *)$.

Raises

- **TypeError** – If *training* is not a bool.
- **TypeError** – If dtype of *eps* or *momentum* is not float.
- **TypeError** – If *input*, *weight*, *bias*, *running_mean* or *running_var* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input_x = Tensor([[1.0, 2.0], [3.0, 4.0]], mindspore.float32)
>>> running_mean = Tensor([0.5, 1.5], mindspore.float32)
>>> running_var = Tensor([0.1, 0.2], mindspore.float32)
>>> weight = Tensor([2.0, 2.0], mindspore.float32)
>>> bias = Tensor([-1.0, -1.0], mindspore.float32)
>>> output = mint.nn.functional.batch_norm(input_x, running_mean, running_var, weight, bias)
>>> print(output)
[[ 2.1621194  1.2360122]
 [14.810596  10.180061 ]]
```

mindspore.mint.nn.functional.elu

`mindspore.mint.nn.functional.`**`elu`**(*input*, *alpha=1.0*, *inplace=False*)

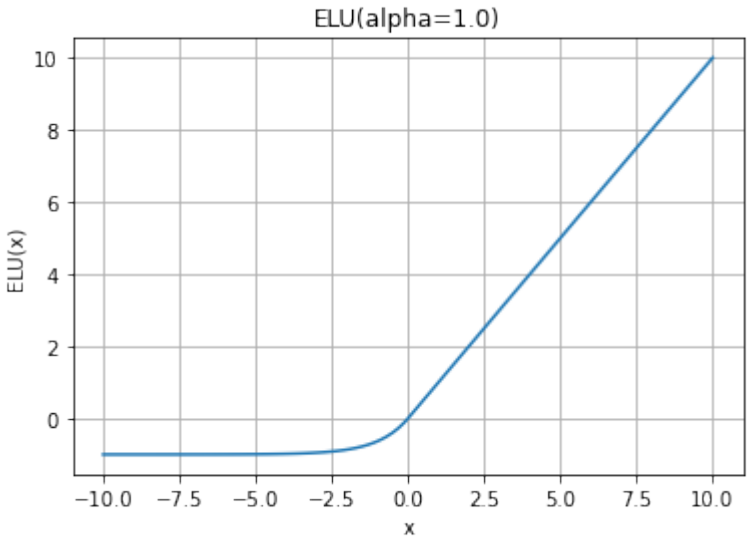
Exponential Linear Unit activation function

Applies the exponential linear unit function element-wise. The activation function is defined as:

$$ELU_i = \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \alpha * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where x_i represents the element of the input and α represents the *alpha* parameter, and *alpha* represents the smoothness of the ELU.

ELU Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (*Tensor*) – The input of ELU is a Tensor of any dimension.
- **alpha** (*float, optional*) – The alpha value of ELU, the data type is float. Default: `1.0`.
- **inplace** (*bool, optional*) – Whether to use inplace mode, the data type is bool. Default: `False`.

Returns

Tensor, with the same shape and type as the *input*.

Raises

- **RuntimeError** – If the dtype of *input* is not float16, float32 or bfloat16.
- **TypeError** – If the dtype of *alpha* is not float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float32)
>>> output = mint.nn.functional.elu(input)
>>> print(output)
[-0.63212055  -0.86466473  0.  2.  1.]
```

mindspore.mint.nn.functional.elu_

mindspore.mint.nn.functional.**elu_**(*input*, *alpha=1.0*)

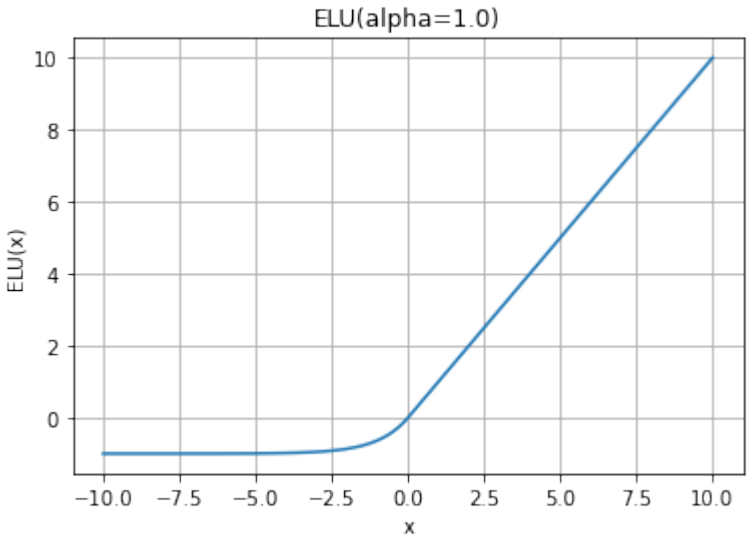
Exponential Linear Unit activation function

Applies the exponential linear unit function inplace element-wise. The activation function is defined as:

$$ELU_i = \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \alpha * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where x_i represents the element of the input and α represents the *alpha* parameter, and *alpha* represents the smoothness of the ELU.

ELU Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) −The input of ELU is a Tensor of any dimension.
- **alpha** (*float, optional*) −The alpha value of ELU, the data type is float and *alpha* should be greater than 0. Default: 1.0.

Returns

Tensor, with the same shape and type as the *input*.

Raises

- **RuntimeError** −If the dtype of *input* is not float16, float32 or bfloat16.
- **TypeError** −If the dtype of *alpha* is not float.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float32)
>>> mint.nn.functional.elu_(input)
>>> print(input)
[-0.63212055  -0.86466473  0.  2.  1.]
```

mindspore.mint.nn.functional.gelu

mindspore.mint.nn.functional.**gelu**(*input*, *, *approximate='none'*) → *Tensor*

Gaussian Error Linear Units activation function.

GeLU is described in the paper Gaussian Error Linear Units (GELUs). And also please refer to BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

When *approximate* argument is *none*, GELU is defined as follows:

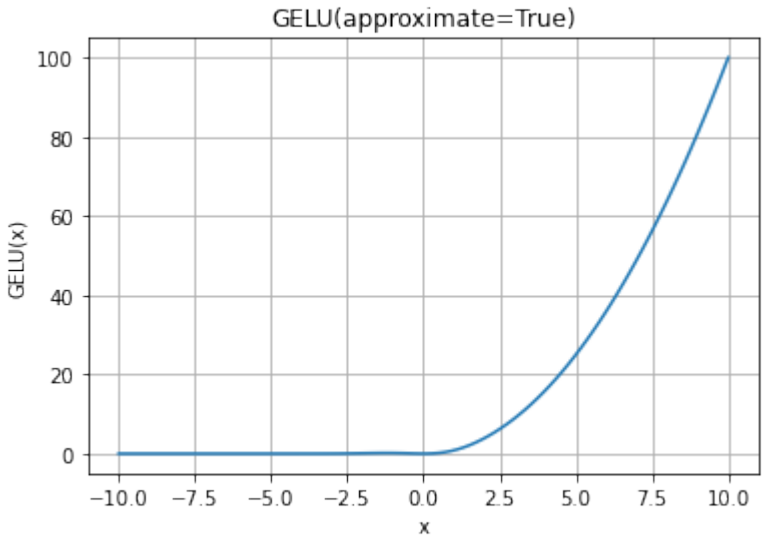
$$GELU(x_i) = x_i * P(X < x_i),$$

where P is the cumulative distribution function of the standard Gaussian distribution, x_i is the input element.

When *approximate* argument is *tanh*, GELU is estimated with:

$$GELU(x_i) = 0.5 * x_i * (1 + \tanh(\sqrt{(2/\pi)} * (x_i + 0.044715 * x_i^3)))$$

GELU Activation Function Graph:



Note: On the Ascend platform, when *input* is -inf, its gradient is 0, and when *input* is inf, its gradient is *dout*.

Parameters

input (Tensor) −The input of the activation function GeLU, the data type is float16, float32 or float64.

Keyword Arguments

approximate (*str, optional*) –the gelu approximation algorithm to use. Acceptable vaslues are `'none'` and `'tanh'`. Default: `'none'`.

Returns

Tensor, with the same type and shape as *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not bfloat16, float16, float32 or float64.
- **ValueError** –If *approximate* value is neither *none* nor *tanh*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> result = mint.nn.functional.gelu(input)
>>> print(result)
[[-1.58655241e-01  3.99987316e+00 -0.00000000e+00]
 [ 1.95449972e+00 -1.41860323e-06  9.0000000e+00]]
>>> result = mint.nn.functional.gelu(input, approximate="tanh")
>>> print(result)
[[-1.58808023e-01  3.99992990e+00 -3.10779147e-21]
 [ 1.95459759e+00 -2.29180174e-07  9.0000000e+00]]
```

mindspore.mint.nn.functional.glu

`mindspore.mint.nn.functional.`**glu**(*input*, *dim=- 1*)

Computes GLU (Gated Linear Unit activation function) of the input tensor.

$$GLU(a, b) = a \otimes \sigma(b)$$

where a is the first half of the *input* Tensor after *input* is split and b is the second half.

Here σ is the sigmoid function, and $\otimes$ is the Hadamard product. See Language Modeling with Gated Convluational Networks.

Parameters

- **input** (`Tensor`) –Tensor to be calculated. Dtype is floating point and the shape is $(*_1, N, *_2)$ where * means, any number of additional dimensions. N is required to be an even number, where N is the size of *input* on the dimension selected by *dim*.
- **dim** (*int, optional*) –The dimension to split the input *input*. The value range is *[-r, r)* where *r* is the number of dimensions of *input*. Default: `-1` , the last dimension in *input*.

Returns

Tensor, the same dtype as the input *input*. The shape is $(*_1, M, *_2)$ where $M = N/2$.

Raises

- **TypeError** –If *input* is not a Tensor or *dim* is not an int.
- **IndexError** –If the value of *dim* is out of the range of *[-r, r)*, where *r* is the number of dimensions of *input*.
- **RuntimeError** –If dtype of *input* is not supported.
- **RuntimeError** –If the length of *input* in the dimension selected by *dim* is not even.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> input = Tensor([[0.1, 0.2, 0.3, 0.4], [0.5, 0.6, 0.7, 0.8]])
>>> output = mint.nn.functional.glu(input)
>>> print(output)
[[0.05744425 0.11973753]
 [0.33409387 0.41398472]]
```

mindspore.mint.nn.functional.group_norm

mindspore.mint.nn.functional.**group_norm**(*input*, *num_groups*, *weight=None*, *bias=None*, *eps=1e-5*)

Group Normalization over a mini-batch of inputs.

Group Normalization is widely used in recurrent neural networks. It applies normalization on a mini-batch of inputs for each single training case as described in the paper Group Normalization. Group Normalization divides the channels into groups and computes within each group the mean and variance for normalization, and it performs very stable over a wide range of batch size. γ and β are trainable scale and shift. It can be described using the following formula:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

where γ is *weight*, β is *bias*, ϵ is *eps*.

Parameters

- **input** (Tensor) –The input feature with shape $(N, C, *)$ where $*$ means, any number of additional dimensions.
- **num_groups** (*int*) –The number of groups to be divided along the channel dimension.
- **weight** (Tensor, *optional*) –The shape $(C,)$, Default: `None`, has the same data type with *input*.
- **bias** (Tensor, *optional*) –The shape $(C,)$, Default: `None`, has the same data type with *input*.
- **eps** (*float*, *optional*) –A value added to the denominator for numerical stability. Default: `1e-5` .

Returns

Tensor, the normalized and scaled offset tensor, has the same shape and data type as the *input*.

Raises

- **TypeError** –If *num_groups* is not an int.
- **TypeError** –If *eps* is not a float.
- **ValueError** –If *num_groups* is less than 1.

- **ValueError** – If *C* (the second parameter of dimensions of *input*) is not divided by *num_groups*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import mint
>>> x = ms.Tensor(np.ones([1, 2, 4, 4], np.float32))
>>> output = mint.nn.functional.group_norm(x, 2)
>>> print(output)
[[[[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]
  [[0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]
   [0. 0. 0. 0.]]]]
```

mindspore.mint.nn.functional.hardshrink

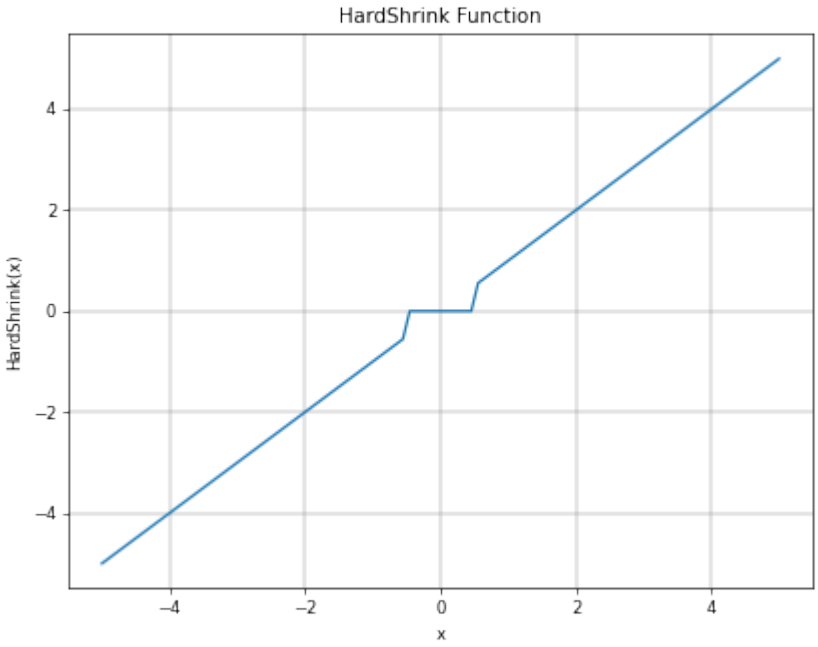
`mindspore.mint.nn.functional.`**`hardshrink`**(*input*, *lambd=0.5*)

Hard Shrink activation function. Calculates the output according to the input elements.

The formula is defined as follows:

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

HardShrink Activation Function Graph:



Parameters

- **input** (Tensor) – The input of Hard Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
- **lambd** (*number, optional*) – The threshold λ defined by the Hard Shrink formula. Default: `0.5`.

Returns

Tensor, has the same data type and shape as the input *input*.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[0.5, 1, 2.0], [0.0533, 0.0776, -2.1233]]), mindspore.float32)
>>> output = mint.nn.functional.hardshrink(input)
>>> print(output)
[[ 0.      1.      2.    ]
 [ 0.      0.     -2.1233]]
```

mindspore.mint.nn.functional.hardsigmoid

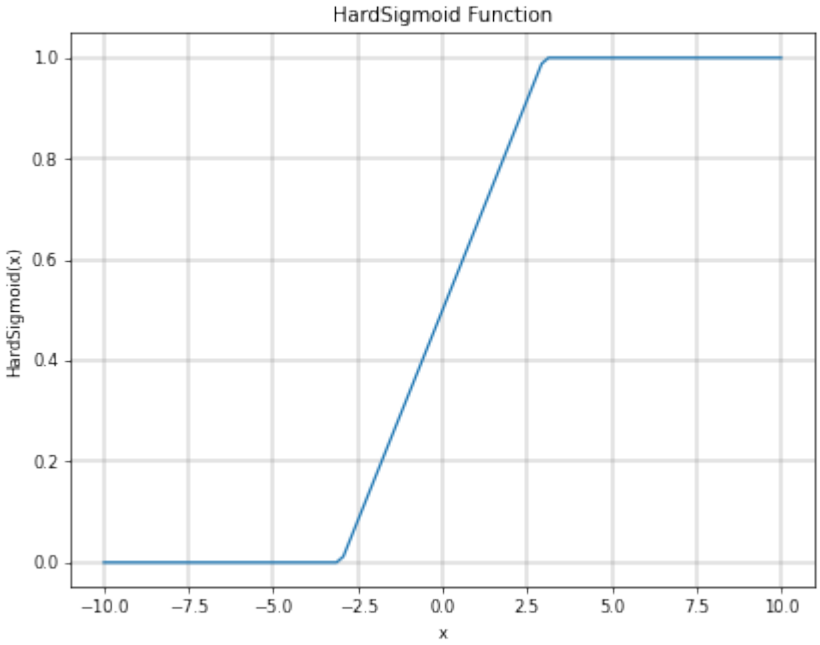
mindspore.mint.nn.functional.**hardsigmoid**(*input*)

Hard Sigmoid activation function. Calculates the output according to the input elements.

Hard Sigmoid is defined as:

$$\text{HardSigmoid}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ 1, & \text{if } input \geq +3, \\ input/6 + 1/2, & \text{otherwise} \end{cases}$$

HardSigmoid Activation Function Graph:



Parameters

input (Tensor) – The input Tensor.

Returns

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *input* is neither int nor float.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> output = mint.nn.functional.hardsigmoid(input)
>>> print(output)
[0.3333 0.1666 0.5    0.8335 0.6665]
```

mindspore.mint.nn.functional.hardswish

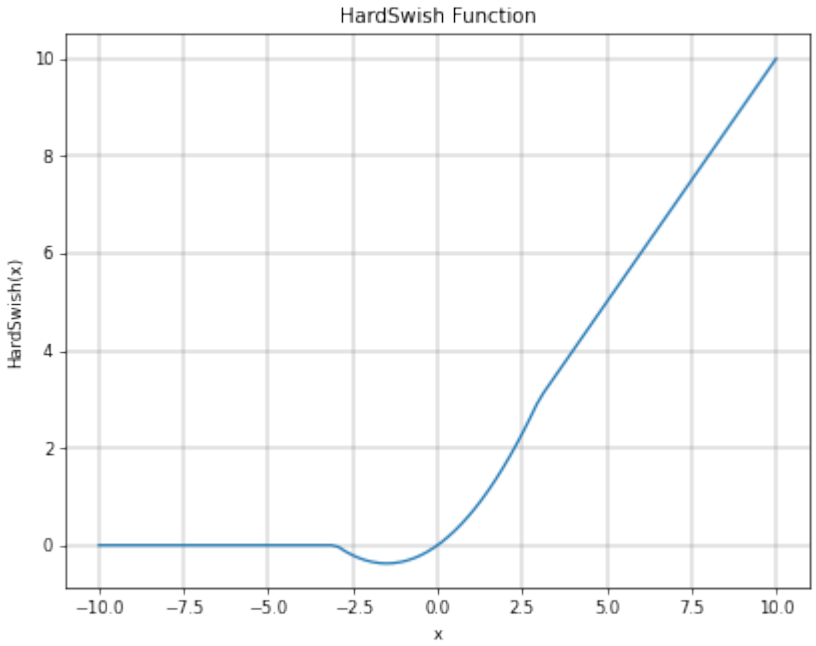
mindspore.mint.nn.functional.**hardswish**(*input*)

Hard Swish activation function. The input is a Tensor with any valid shape.

Hard swish is defined as:

$$\text{HardSwish}(input) = \begin{cases} 0, & \text{if } input \leq -3, \\ input, & \text{if } input \geq +3, \\ input * (input + 3)/6, & \text{otherwise} \end{cases}$$

HardSwish Activation Function Graph:



Parameters

input (Tensor) – The input Tensor.

Returns

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *input* is neither int nor float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([-1, -2, 0, 2, 1]), mindspore.float16)
>>> output = mint.nn.functional.hardswish(input)
>>> print(output)
[-0.3333  -0.3333  0  1.667  0.6665]
```

mindspore.mint.nn.functional.layer_norm

`mindspore.mint.nn.functional.`**`layer_norm`**(*input*, *normalized_shape*, *weight=None*, *bias=None*, *eps=1e-5*)

Applies the Layer Normalization on the mini-batch input.

Layer normalization is widely used in recurrent neural networks. Apply normalization to the mini-batch input of a single training case. LayerNorm is described in the paper Layer Normalization.

Unlike batch normalization, layer normalization performs the exact same calculations at training and test time. Applies to all channels and pixels, even batch_size=1. The formula is as follows:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

where γ is the weight value learned through training, β is the bias value learned through training.

Parameters

- **input** (`Tensor`) –The shape of input is *(N, *)*, where * represents any additional dimension.
- **normalized_shape** (*`Union(int, tuple[int], list[int])`*) –The normalized shape of *input* for LayerNorm. *normalized_shape* equal to *input_shape[begin_norm_axis:]*, where *begin_norm_axis* represents the axis where normalization begins.
- **weight** (`Tensor, optional`) –Learnable parameter γ . Tensor of shape *normalized_shape*. Default: `None`, has the same data type with *input*. Initialized to `1` when *weight* is None.
- **bias** (`Tensor, optional`) –Learnable parameter β . Tensor of shape *normalized_shape*. Default: `None`, has the same data type with *input*. Initialized to `0` when *bias* is None.
- **eps** (*`float, optional`*) –A value added to the denominator for numerical stability(ϵ). Default: `1e-5` .

Returns

Tensor. The normalized tensor, has the same type and shape as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *normalized_shape* is not an integer, a list or a tuple.
- **TypeError** –If *eps* is not a float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.array([[1, 2, 3], [1, 2, 3]]), mindspore.float32)
>>> normalized_shape = (3,)
>>> gamma = Tensor(np.ones(normalized_shape), mindspore.float32)
>>> beta = Tensor(np.zeros(normalized_shape), mindspore.float32)
>>> eps = 1e-7
>>> output = mint.nn.functional.layer_norm(input_x, normalized_shape, gamma, beta, eps)
>>> print(output)
[[-1.2247448 0. 1.2247448]
 [-1.2247448 0. 1.2247448]]
```

mindspore.mint.nn.functional.leaky_relu

`mindspore.mint.nn.functional.`**`leaky_relu`**(*input*, *negative_slope=0.01*)

leaky_relu activation function. The element of *input* less than 0 times *negative_slope* .

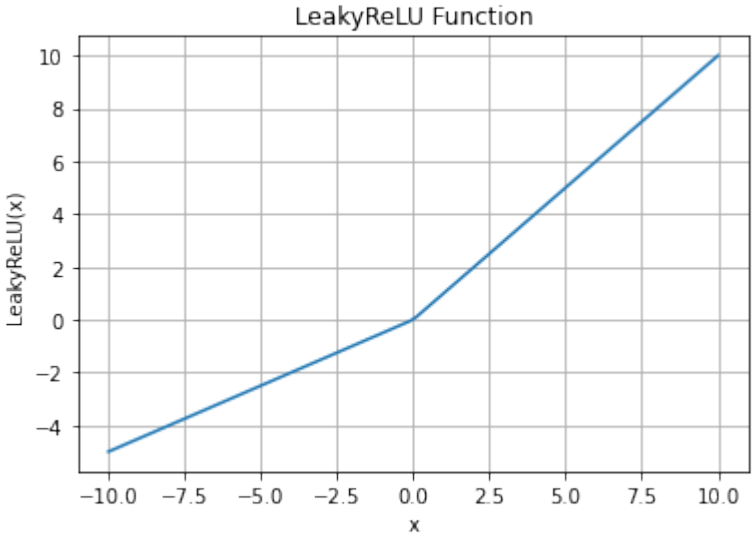
The activation function is defined as:

$$\text{leaky\_relu}(input) = \begin{cases} input, & \text{if } input \geq 0; \\ \text{negative\_slope} * input, & \text{otherwise.} \end{cases}$$

where $negative\_slope$ represents the *negative_slope* parameter.

For more details, see Rectifier Nonlinearities Improve Neural Network Acoustic Models.

LeakyReLU Activation Function Graph:



Parameters

- **`input`** (`Tensor`) –The input of leaky_relu is a Tensor of any dimension.

- **negative_slope** (*Union[int, float], optional*) –Slope of the activation function when the element of *input* is less than 0. Default: `0.01` .

Returns

Tensor, has the same type and shape as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *negative_slope* is not a float or an int.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> print(mint.nn.functional.leaky_relu(input, negative_slope=0.2))
[[-0.2  4.  -1.6]
 [ 2.  -1.   9. ]]
```

mindspore.mint.nn.functional.log_softmax

`mindspore.mint.nn.functional.`**log_softmax**(*input*, *dim=None*, *, *dtype=None*)

Applies the Log Softmax function to the input tensor on the specified axis. Supposes a slice in the given axis, x for each element x_i, the Log Softmax function is shown as follows:

$$\text{output}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)}\right),$$

where N is the length of the Tensor.

Parameters

- **input** (`Tensor`) –The input Tensor.
- **dim** (*int, optional*) –The axis to perform the Log softmax operation. Default: `None` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –The desired dtype of returned Tensor. If not set to None, the input Tensor will be cast to *dtype* before the operation is performed. This is useful for preventing overflows. If set to None, stay the same as original Tensor. Default: `None` .

Returns

Tensor, with the same shape as the input.

Raises

- **TypeError** –If *dim* is not an int.
- **ValueError** –If *dim* is not in range [-len(input.shape), len(input.shape)).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> logits = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.nn.functional.log_softmax(logits, dim=-1)
>>> print(output)
[-4.4519143 -3.4519143 -2.4519143 -1.4519144 -0.4519144]
```

mindspore.mint.nn.functional.logsigmoid

`mindspore.mint.nn.functional.`**`logsigmoid`**(*input*)

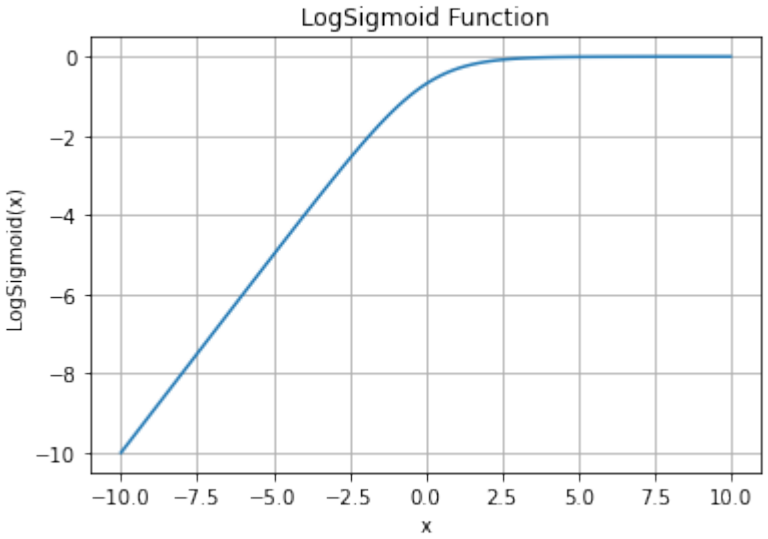
Applies logsigmoid activation element-wise. The input is a Tensor with any valid shape.

Logsigmoid is defined as:

$$\text{logsigmoid}(x_i) = \log(\frac{1}{1 + \exp(-x_i)}),$$

where x_i is the element of the input.

LogSigmoid Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Parameters

`input` (`Tensor`) −The input of LogSigmoid with data type of bfloat16, float16 or float32. The shape is $(*)$ where $*$ means, any number of additional dimensions.

Returns

Tensor, with the same type and shape as the *input*.

Raises

- **TypeError** – If dtype of *input* is not bfloat16, float16 and float32.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([1.0, 2.0, 3.0], mindspore.float32)
>>> output = mint.nn.functional.logsigmoid(input)
>>> print(output)
[-0.31326166 -0.12692806 -0.04858734]
```

mindspore.mint.nn.functional.mish

mindspore.mint.nn.functional.**mish**(*input*)

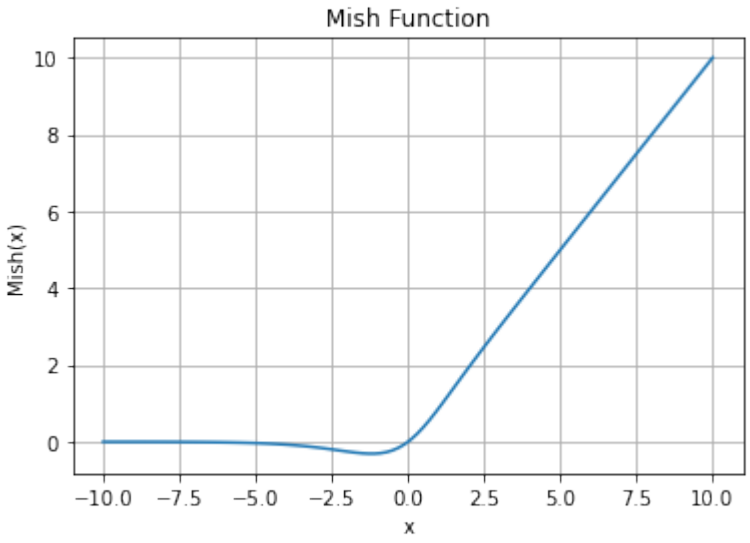
Computes MISH (A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.

The formula is defined as follows:

$$\text{mish}(input) = input * \tanh(softplus(\text{input}))$$

See more details in A Self Regularized Non-Monotonic Neural Activation Function.

Mish Activation Function Graph:



Parameters

input (Tensor) – The input of MISH. Supported dtypes:

- Ascend: float16, float32.

Returns

Tensor, has the same type and shape as the *input*.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If dtype of *input* is not float16 or float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> x = Tensor(np.array([[-1.1, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = mint.nn.functional.mish(x)
>>> print(output)
[[-3.0764845e-01 3.9974124e+00 -2.6832507e-03]
 [ 1.9439589e+00 -3.3576239e-02 8.9999990e+00]]
```

mindspore.mint.nn.functional.prelu

mindspore.mint.nn.functional.**prelu**(*input*, *weight*)

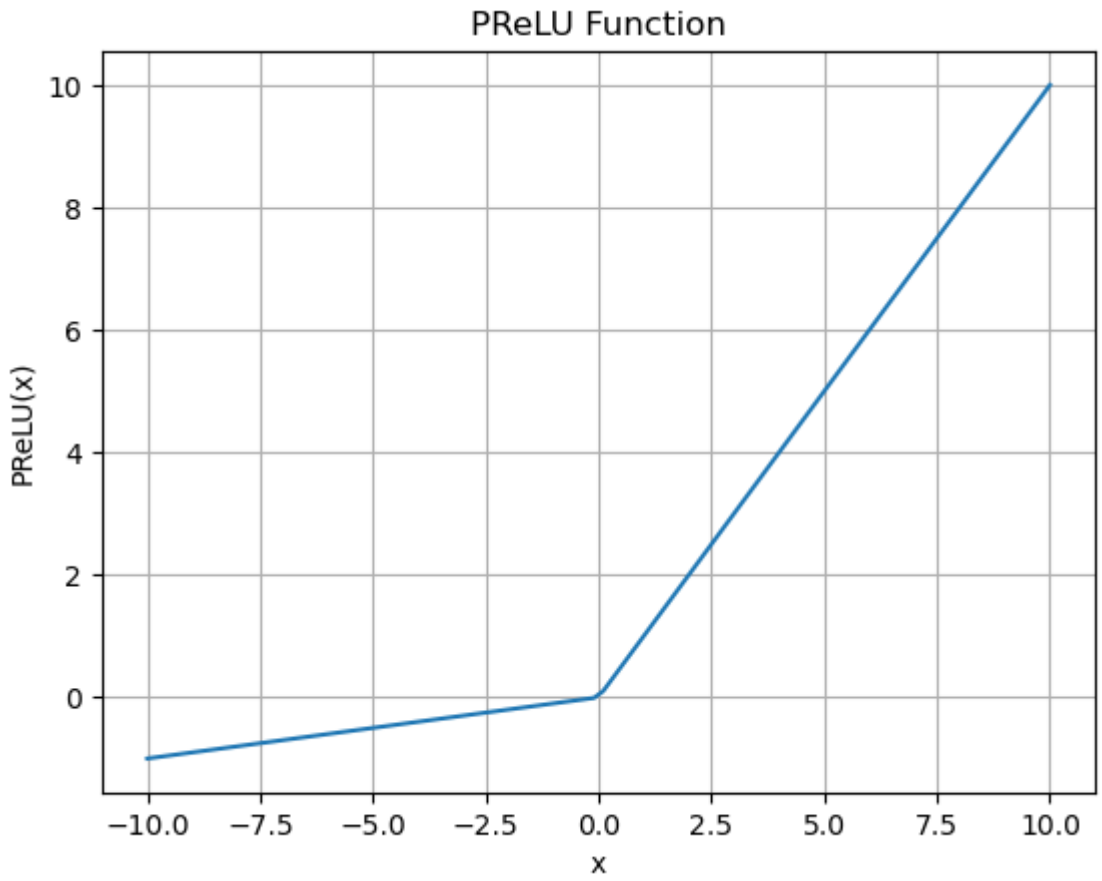
Parametric Rectified Linear Unit activation function.

PReLU is described in the paper Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Defined as follows:

$$prelu(x_i) = \max(0, x_i) + \min(0, w * x_i),$$

where x_i is an element of a channel of the input, w is the weight of the channel.

PReLU Activation Function Graph:



Note: Channel dim is the 2nd dim of input. When input has dims < 2, then there is no channel dim and the number of channels = 1.

Parameters

- **input** (Tensor) −The input Tensor of the activation function.
- **weight** (Tensor) −Weight Tensor. The size of the weight should be 1 or the number of channels at Tensor *input*.

Returns

Tensor, with the same shape and dtype as *input*. For detailed information, please refer to *mindspore.mint.nn.PReLU*.

Raises

- **TypeError** −If the *input* or the *weight* is not a Tensor.
- **ValueError** −If the *weight* is not a 0-D or 1-D Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.arange(-6, 6).reshape((2, 3, 2)), mindspore.float32)
>>> weight = Tensor(np.array([0.1, 0.6, -0.3]), mindspore.float32)
>>> output = mint.nn.functional.prelu(x, weight)
>>> print(output)
[[[-0.60 -0.50]
  [-2.40 -1.80]
  [ 0.60  0.30]]
 [[ 0.00  1.00]
  [ 2.00  3.00]
  [ 4.0   5.00]]]
```

mindspore.mint.nn.functional.relu

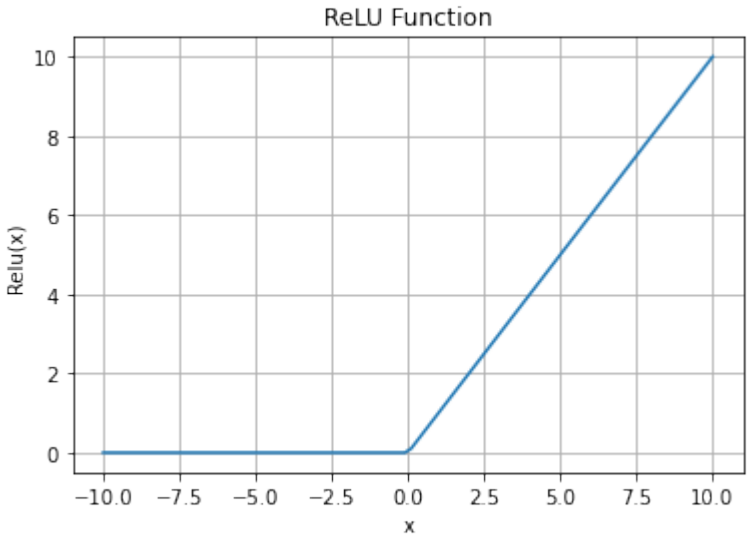
mindspore.mint.nn.functional.**relu**(*input*, *inplace=False*)

Computes ReLU (Rectified Linear Unit activation function) of input tensors element-wise.

It returns $\max(input,\ 0)$ element-wise. Specially, the neurons with the negative output will be suppressed and the active neurons will stay the same.

$$ReLU(input) = (input)^+ = \max(0, input)$$

ReLU Activation Function Graph:



Parameters

- **input** (Tensor) –The input Tensor.
- **inplace** (*bool, optional*) –Whether to use inplace mode, Defaults to False.

Returns

Tensor, with the same dtype and shape as the *input*.

Raises

- **TypeError** – If dtype of *input* is not Number type.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = mint.nn.functional.relu(input)
>>> print(output)
[[0. 4. 0.]
 [2. 0. 9.]]
```

mindspore.mint.nn.functional.relu6

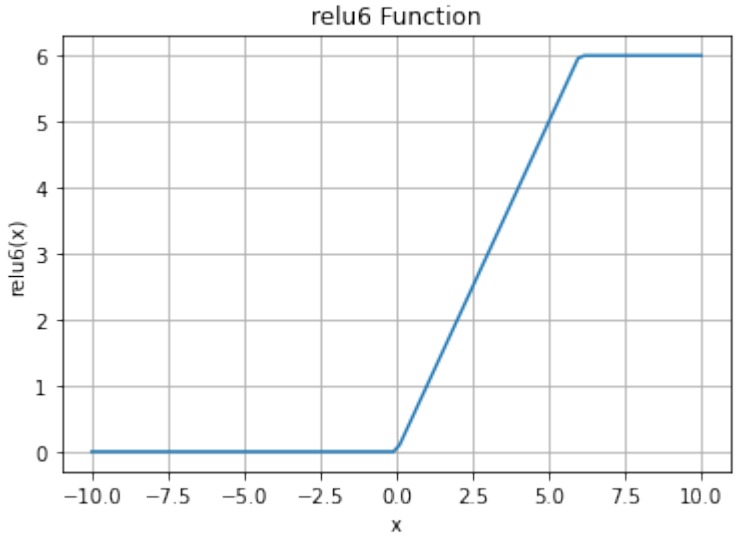
`mindspore.mint.nn.functional.`**`relu6`**(*input*, *inplace=False*)

Computes ReLU (Rectified Linear Unit) upper bounded by 6 of input tensors element-wise.

$$\text{ReLU6}(input) = \min(\max(0, input), 6)$$

It returns $\min(\max(0, input), 6)$ element-wise.

ReLU6 Activation Function Graph:



Warning: This is an experimental optimizer API that is subject to change.

Parameters

- **input** (`Tensor`) – input Tensor. Dtype is in int8, int16, int32, int64, uint8, float16, float32, bfloat16.

- **inplace** (*bool, optional*) −Whether to apply erasing inplace. Default: `False`.

Returns

Tensor, with the same dtype and shape as the *input*.

Raises

- **TypeError** −If *input* is not a Tensor.
- **TypeError** −If dtype of *input* is not one of: int8, int16, int32, int64, uint8, float16, float32, bfloat16.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> result = mint.nn.functional.relu6(x)
>>> print(result)
[[0. 4. 0.]
 [2. 0. 6.]]
```

mindspore.mint.nn.functional.relu_

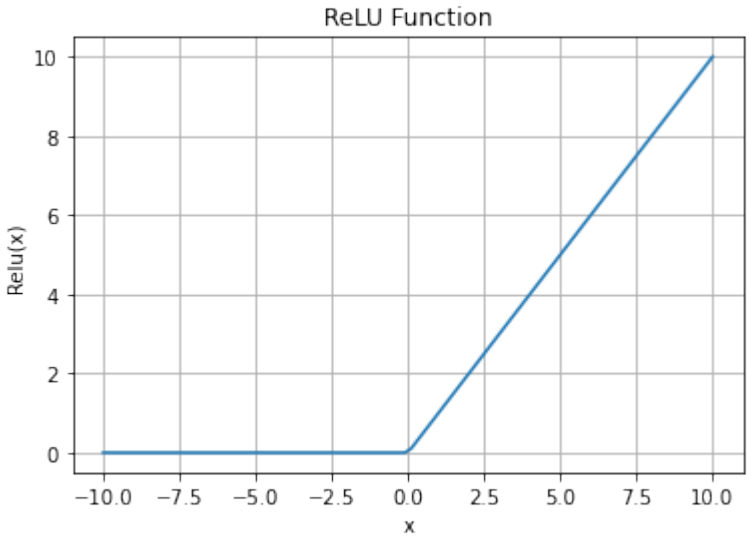
mindspore.mint.nn.functional.**relu_**(*input*)

ReLuComputes ReLU (Rectified Linear Unit activation function) inplace of input tensors element-wise.

It returns $\max(input,\ 0)$ element-wise. Specially, the neurons with the negative output will be suppressed and the active neurons will stay the same.

$$ReLU(input) = (input)^+ = \max(0, input)$$

ReLU Activation Function Graph:



Warning: This is an experimental API that is subject to change or deletion.

Parameters

input (Tensor) –The input Tensor.

Returns

Tensor, with the same dtype and shape as the *input*.

Raises

- **TypeError** –If dtype of *input* is not Number type.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> mint.nn.functional.relu_(input)
>>> print(input)
[[0. 4. 0.]
 [2. 0. 9.]]
```

mindspore.mint.nn.functional.selu

mindspore.mint.nn.functional.**selu**(*input*)

Activation function SELU (Scaled exponential Linear Unit).

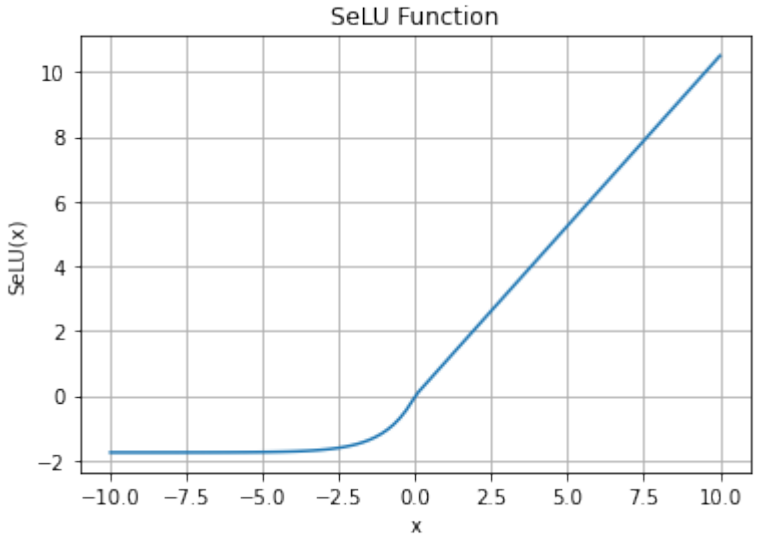
The activation function is defined as:

$$E_i = scale * \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \text{alpha} * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where $alpha$ and $scale$ are pre-defined constants($alpha = 1.67326324$ and $scale = 1.05070098$).

See more details in Self-Normalizing Neural Networks.

SELU Activation Function Graph:



Parameters

input (Tensor) – Tensor of any dimension. The data type is float16, float32, bfloat16.

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is not float16, float32, bfloat16.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> output = mint.nn.functional.selu(input)
>>> print(output)
[[-1.1113307 4.202804 -1.7575096]
 [ 2.101402 -1.7462534 9.456309 ]]
```

mindspore.mint.nn.functional.sigmoid

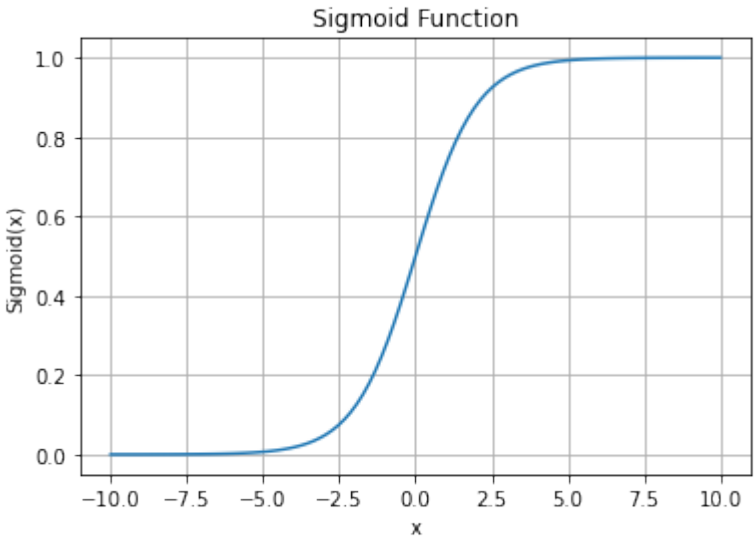
mindspore.mint.nn.functional.**sigmoid**(*input*)

Computes Sigmoid of input element-wise. The Sigmoid function is defined as:

$$\text{sigmoid}(x_i) = \frac{1}{1 + \exp(-x_i)}$$

where x_i is an element of x.

Sigmoid Function Graph:



Parameters

input (Tensor) – *input* is *x* in the preceding formula. Tensor of any dimension, the data type is float16, float32, float64, complex64 or complex128.

Returns

Tensor, with the same type and shape as the input.

Raises

- **TypeError** – If dtype of *input* is not float16, float32, float64, complex64 or complex128.
- **TypeError** – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.nn.functional.sigmoid(input)
>>> print(output)
[0.7310586  0.880797   0.95257413 0.98201376 0.9933072 ]
```

mindspore.mint.nn.functional.silu

mindspore.mint.nn.functional.**silu**(*input*)

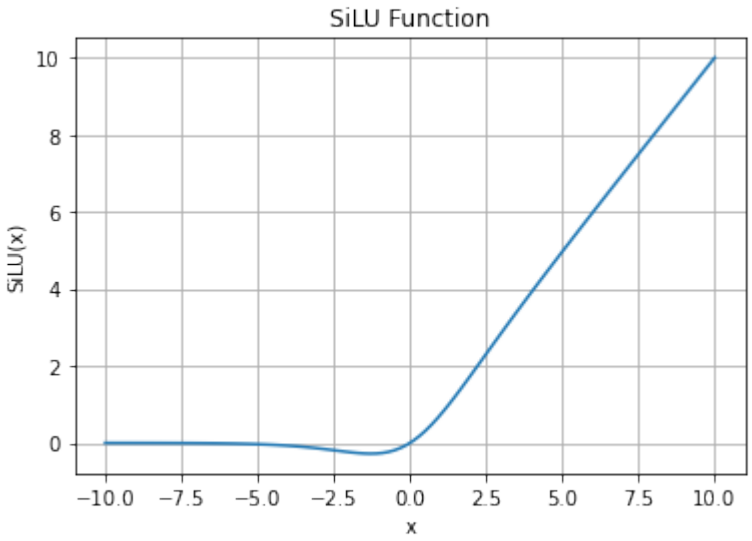
Computes Sigmoid Linear Unit of input element-wise, also known as Swish function. The SiLU function is defined as:

$$\text{SiLU}(x) = x * \sigma(x),$$

where x is an element of the input, $\sigma(x)$ is Sigmoid function.

$$\text{sigma}(x_i) = \frac{1}{1 + \exp(-x_i)},$$

SiLU Function Graph:



Parameters

input (Tensor) – *input* is x in the preceding formula. Input with the data type float16 or float32.

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If dtype of *input* is neither float16 nor float32.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> import numpy as np
>>> input = Tensor(np.array([-1, 2, -3, 2, -1]), mindspore.float16)
>>> output = mint.nn.functional.silu(input)
>>> print(output)
[-0.269  1.762  -0.1423  1.762  -0.269]
```

mindspore.mint.nn.functional.softmax

mindspore.mint.nn.functional.**softmax**(*input*, *dim=None*, *dtype=None*)

Applies the Softmax operation to the input tensor on the specified axis. Suppose a slice in the given axis dim, then for each element $input_i$, the Softmax function is shown as follows:

$$\text{output}(input_i) = \frac{\exp(input_i)}{\sum_{j=0}^{N-1}\exp(input_j)},$$

where N is the length of the tensor.

Parameters

- **input** (Tensor) –Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions.
- **dim** (*int, optional*) –The dim to perform the Softmax operation. Default: None .
- **dtype** (*mindspore.dtype*, optional) –When set, *input* will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. Default: None .

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError –If *dim* is not an int.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.nn.functional.softmax(input)
>>> print(output)
[0.01165623 0.03168492 0.08612854 0.23412167 0.6364086 ]
```

mindspore.mint.nn.functional.softplus

mindspore.mint.nn.functional.**softplus**(*input*, *beta=1*, *threshold=20*)

Applies softplus function to *input* element-wise.

The softplus function is shown as follows, x is the element of *input* :

$$\text{output} = \frac{1}{beta}\log(1 + \exp(\text{beta} * \text{x}))$$

where $input * beta > threshold$, the implementation converts to the linear function to ensure numerical stability.

Parameters

- **input** (Tensor) –Tensor of any dimension. Supported dtypes:
  - Ascend: float16, float32, bfloat16.

- **beta** (*number.Number, optional*) –Scaling parameters in the softplus function. Default: `1` .
- **threshold** (*number.Number, optional*) –For numerical stability, the softplus function is converted to a threshold parameter of a linear function. Default: `20` .

Returns

Tensor, with the same type and shape as the input.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If dtype of *input* is not float16, float32, bfloat16.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0.1, 0.2, 30, 25]), mindspore.float32)
>>> output = mint.nn.functional.softplus(input)
>>> print(output)
[0.74439657 0.7981388 30. 25.]
```

mindspore.mint.nn.functional.softshrink

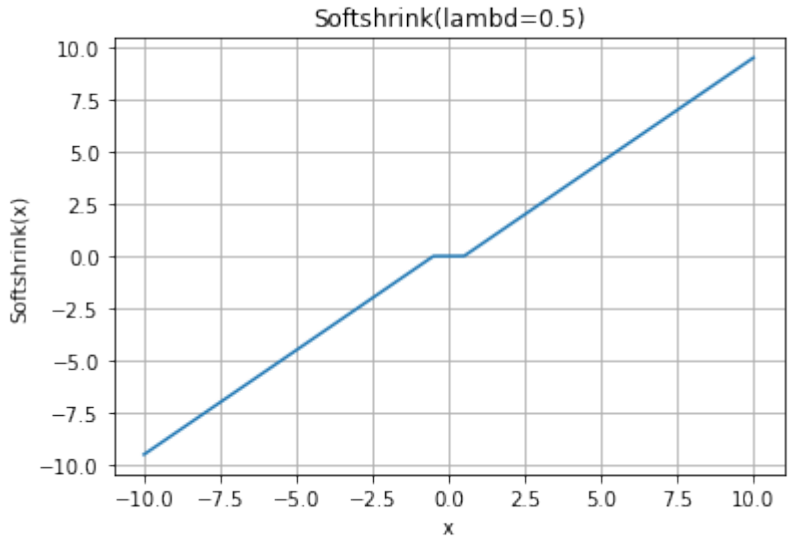
mindspore.mint.nn.functional.**softshrink**(*input*, *lambd=0.5*)

Soft Shrink activation function. Calculates the output according to the input elements.

The formula is defined as follows:

$$\text{SoftShrink}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

SoftShrink Activation Function Graph:



Parameters

- **input** (Tensor) – The input of Soft Shrink. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
- **lambd** (*number, optional*) – The threshold λ defined by the Soft Shrink formula. It should be greater than or equal to 0, default: `0.5`.

Returns

Tensor, has the same data type and shape as the input *input*.

Raises

- **TypeError** – If *lambd* is not a float, int or bool.
- **TypeError** – If *input* is not a tensor.
- **TypeError** – If dtype of *input* is not float16, float32 or bfloat16.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore import mint
>>> import numpy as np
>>> x = Tensor(np.array([[ 0.5297,  0.7871,  1.1754], [ 0.7836,  0.6218, -1.1542]]),␣
↪mindspore.float32)
>>> output = mint.nn.functional.softshrink(x)
>>> print(output)
[[ 0.02979  0.287    0.676  ]
[ 0.2837   0.1216  -0.6543 ]]
```

mindspore.mint.nn.functional.tanh

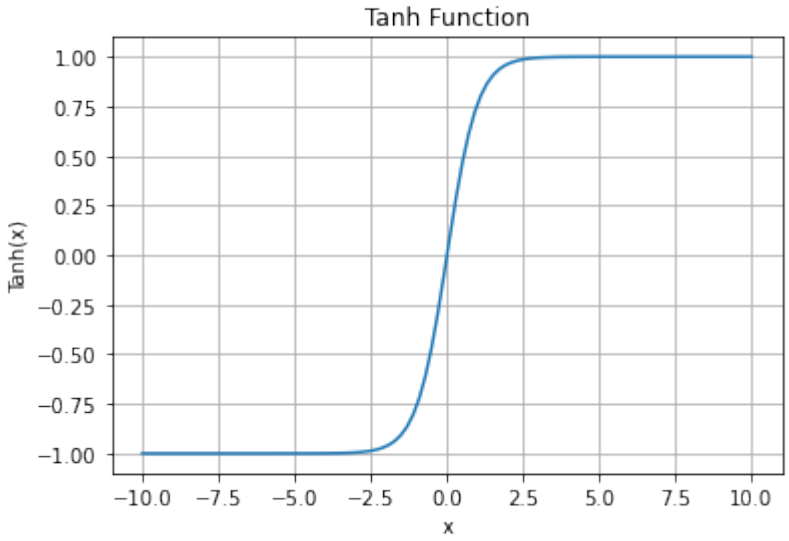
mindspore.mint.nn.functional.**tanh**(*input*)

Computes hyperbolic tangent of input element-wise. The Tanh function is defined as:

$$tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)} = \frac{\exp(2x_i) - 1}{\exp(2x_i) + 1},$$

where x_i is an element of the input Tensor.

Tanh Activation Function Graph:



Parameters

input (Tensor) – Input of Tanh.

Returns

Tensor, with the same type and shape as the *input*.

Raises

TypeError – If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.nn.functional.tanh(input)
>>> print(output)
[0.7615941 0.9640276 0.9950547 0.9993293 0.9999092]
```

4.5.4 Normalization functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.normalize	Perform normalization of inputs over specified dimension	Ascend

mindspore.mint.nn.functional.normalize

mindspore.mint.nn.functional.**normalize**(*input*, *p=2.0*, *dim=1*, *eps=1e-12*)

Perform normalization of inputs over specified dimension

For a tensor input of sizes $(n_0, ..., n_{dim}, ..., n_k)$, each n_{dim} -element vector v along dimension *dim* is transformed as

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}$$

With the default arguments it uses the Euclidean norm over vectors along dimension 1 for normalization.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) – input tensor of any shape.
- **p** (*float*) – the exponent value in the norm formulation. default: 2.
- **dim** (*int*) – the dimension to reduce. default: 1.
- **eps** (*float*) – small value to avoid division by zero. default: 1e-12.

Returns

Tensor, shape and data type are the same as input.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> tensor = Tensor(np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]]), mindspore.float32)
>>> output = mint.nn.functional.normalize(tensor)
>>> print(output)
[[0.0000 0.4472 0.8944]
 [0.4243 0.5657 0.7071]
 [0.4915 0.5735 0.6554]]
```

4.5.5 Linear functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.linear	Applies the dense connected operation to the *input*.	Ascend

mindspore.mint.nn.functional.linear

mindspore.mint.nn.functional.**linear**(*input*, *weight*, *bias=None*)

Applies the dense connected operation to the *input*. The dense function is defined as:

$$output = input * weight^T + bias$$

Warning:

- This is an experimental API that is subject to change or deletion.
- On the Ascend platform, if *bias* is not 1D, the *input* cannot be greater than 6D in PYNATIVE or KBK mode.

Parameters

- **input** (Tensor) –Input Tensor of shape $(*, in\_channels)$, where $*$ means any number of additional dimensions.
- **weight** (Tensor) –The weight applied to the input. The shape is $(out\_channels, in\_channels)$ or $(in\_channels)$.
- **bias** (Tensor, *optional*) –Additive biases to the output. The shape is $(out\_channels)$ or (). Defaults: None, the *bias* is 0.

Returns

Output whose shape is determined by the shape of the input and the weight.

Raises

- **TypeError** –If *input* is not Tensor.
- **TypeError** –If *weight* is not Tensor.
- **TypeError** –If *bias* is not Tensor.
- **RuntimeError** –On the Ascend platform, if *bias* is not 1D and *input* is greater than 6D in PYNATIVE or KBK mode.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[-1., 1., 2.], [-3., -3., 1.]], mindspore.float32)
>>> weight = Tensor([[-2., -2., -2.], [0., -1., 0.]], mindspore.float32)
>>> bias = Tensor([0., 1.], mindspore.float32)
>>> output = mint.nn.functional.linear(input, weight, bias)
>>> print(output)
[[-4.  0.]
 [10.  4.]]
```

4.5.6 Dropout functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.dropout	During training, randomly zeroes some of the elements of the input tensor with probability *p* from a Bernoulli distribution.	Ascend
mindspore.mint.nn.functional.dropout2d	During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of (N, C, H, W), the channel feature map refers to a 2-dimensional feature map with the shape of (H, W)).	Ascend

mindspore.mint.nn.functional.dropout

mindspore.mint.nn.functional.**dropout**(*input*, *p=0.5*, *training=True*, *inplace=False*)

During training, randomly zeroes some of the elements of the input tensor with probability *p* from a Bernoulli distribution. It plays the role of reducing neuron correlation and avoid overfitting. And the return will be multiplied by $\frac{1}{1-p}$ during training. During the reasoning, this operation returns the same Tensor as the *input*.

Parameters

- **input** (Tensor) –The input Tensor of shape $(*, N)$.
- **p** (*float, optional*) –The dropping rate of input neurons, between 0 and 1, e.g. $p = 0.1$, means dropping out 10% of input neurons. Default: `0.5` .
- **training** (*bool, optional*) –Apply dropout if it is `True` , if it is `False` , the input is returned directly, and *p* is invalid. Default: `True` .
- **inplace** (*bool, optional*) –If set to `True` , will do this operation in-place. Default: `False` .

Returns

- **output** (Tensor) - Zeroed tensor, with the same shape and data type as *input*.

Raises

- **TypeError** –If *p* is not a float.
- **TypeError** –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor(((20, 16), (50, 50)), mindspore.float32)
>>> output = mint.nn.functional.dropout(input, p=0.5)
>>> print(output.shape)
(2, 2)
```

mindspore.mint.nn.functional.dropout2d

mindspore.mint.nn.functional.**dropout2d**(*input*, *p=0.5*, *training=True*)

During training, randomly zeroes some channels of the input tensor with probability *p* from a Bernoulli distribution (For a 4-dimensional tensor with a shape of (N, C, H, W), the channel feature map refers to a 2-dimensional feature map with the shape of (H, W)).

For example, the *j_th* channel of the *i_th* sample in the batched input is a to-be-processed *2D* tensor input[i,j]. Each channel will be zeroed out independently on every forward call which based on Bernoulli distribution probability *p*. The parper Dropout: A Simple Way to Prevent Neural Networks from Overfitting mentioned this technology, and it is proved that it can effectively reduce over fitting and prevent neuronal coadaptation. For more details, refer to Improving neural networks by preventing co-adaptation of feature detectors .

dropout2d can improve the independence between channel feature maps.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –A *4D* tensor with shape (N, C, H, W), where N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.
- **p** (*float, optional*) –The dropping probability of a channel, between 0 and 1, e.g. $p = 0.8$, which means dropping out 80% of channels. Default: 0.5 .
- **training** (*bool, optional*) –If *training* is True, applying dropout, otherwise, not applying. Default: True .

Returns

Tensor, output, with the same shape and data type as *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If the data type of *p* is not float.
- **ValueError** –If *p* is out of the range *[0.0, 1.0]*.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.ones([2, 1, 2, 3]), mindspore.float32)
>>> output = mint.nn.functional.dropout2d(input, 0.5)
>>> print(output.shape)
(2, 1, 2, 3)
```

4.5.7 Sparse functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.embedding	Retrieve the word embeddings in *weight* using indices specified in *input*.	Ascend
mindspore.mint.nn.functional.one_hot	Computes a one-hot tensor.	Ascend

mindspore.mint.nn.functional.embedding

mindspore.mint.nn.functional.**embedding**(*input*, *weight*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*)

Retrieve the word embeddings in *weight* using indices specified in *input*.

> **Warning:** On Ascend, the behavior is unpredictable when the value of input is invalid.

Parameters

- **input** (*Tensor*) – The indices used to lookup in the *weight*. The data type must be mindspore.int32 or mindspore.int64, and the value should be in range *[0, weight.shape[0])*.
- **weight** (*Union[Parameter, Tensor]*) – The matrix where to lookup from. The shape must be 2D.
- **padding_idx** (*int, optional*) – If the value is not None, the corresponding row of *weight* will not be updated in training. The value should be in range *[-weight.shape[0], weight.shape[0])* if it's not None. Default None.
- **max_norm** (*float, optional*) – If not None, firstly get the p-norm result of the *weight* specified by *input* where p is specified by *norm_type*; if the result is larger then *max_norm*, update the *weight* with $\frac{max\_norm}{result+1e^{-7}}$ in-place. Default None.
- **norm_type** (*float, optional*) – Indicates the value of p in p-norm. Default 2.0.
- **scale_grad_by_freq** (*bool, optional*) – If True the gradients will be scaled by the inverse of frequency of the index in *input*. Default False.

Returns

Tensor, has the same data type as *weight*, the shape is $(*input.shape, weight.shape[1])$.

Raises

- **ValueError** – If *padding_idx* is out of valid range.
- **ValueError** – If the shape of *weight* is invalid.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, mint
>>> input = Tensor([[1, 0, 1, 1], [0, 0, 1, 0]])
>>> weight = Parameter(np.random.randn(3, 3).astype(np.float32))
>>> output = mint.nn.functional.embedding(input, weight, max_norm=0.4)
>>> print(output)
[[[ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01]],
 [[ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01],
  [ 5.49015924e-02,  3.47811311e-01, -1.89771220e-01],
  [ 2.09307984e-01, -2.24846993e-02,  3.40124398e-01]]]
```

mindspore.mint.nn.functional.one_hot

mindspore.mint.nn.functional.**one_hot**(*tensor*, *num_classes=- 1*)

Computes a one-hot tensor.

The locations represented by tensor in *tensor* take value *1*, while all other locations take value *0*.

Parameters

- **tensor** (Tensor) – A tensor of indices. Tensor of shape $(X_0, \ldots, X_n)$. Data type must be int32 or int64. Dimension cannot be greater than 7.
- **num_classes** (*int, optional*) – A scalar defining the depth of the one-hot dimension, default: -1.

Returns

Tensor, one-hot tensor.

Raises

- **TypeError** – If *num_classes* is not an int.
- **TypeError** – If dtype of *tensor* is not int32 or int64.
- **ValueError** – If *num_classes* is less than -1.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> tensor = Tensor(np.array([0, 1, 2]), mindspore.int32)
>>> num_classes = 3
>>> output = mint.nn.functional.one_hot(tensor, num_classes)
>>> print(output)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

4.5.8 Loss Functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.cross_entropy	The cross entropy loss between input and target.	Ascend
mindspore.mint.nn.functional.binary_cross_entropy	Computes the binary cross entropy(Measure the difference information between two probability distributions) between predictive value *input* and target value *target*.	Ascend
mindspore.mint.nn.functional.binary _cross_entropy_with_logits	Adds sigmoid activation function to *input* as logits, and uses this logits to compute binary cross entropy between the logits and the target.	Ascend
mindspore.mint.nn.functional.kl_div	Computes the Kullback-Leibler divergence between the *input* and the *target*.	Ascend
mindspore.mint.nn.functional.l1_loss	Calculate the mean absolute error between the *input* value and the *target* value.	Ascend
mindspore.mint.nn.functional.mse_loss	Calculates the mean squared error between the predicted value and the label value.	Ascend
mindspore.mint.nn.functional.nll_loss	Gets the negative log likelihood loss between input and target.	Ascend
mindspore.mint.nn.functional.smooth_l1_loss	Computes smooth L1 loss, a robust L1 loss.	Ascend

mindspore.mint.nn.functional.cross_entropy

mindspore.mint.nn.functional.**cross_entropy**(*input*, *target*, *weight=None*, *ignore_index=- 100*, *reduction='mean'*, *label_smoothing=0.0*)

The cross entropy loss between input and target.

The cross entropy supports two kind of targets:

- Class indices (int) in the range $[0, C)$ where C is the number of classes, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}$$

where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n} \cdot \mathbb{1}\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

- Probabilities (float) for each class, useful when labels beyond a single class per minibatch item are required, the loss with reduction=none can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{\exp(x_{n,c})}{\sum_{i=1}^{C} \exp(x_{n,i})} y_{n,c}$$

where x is the inputs, y is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `None` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \frac{\sum_{n=1}^{N} l_n}{N}, & \text{if reduction = 'mean',} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'.} \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Note: Dynamic shape, dynamic rank and variable constant input are not supported in strict graph mode (jit_syntax_level=mindspore.STRICT).

Parameters

- **input** (Tensor) –(N) or (N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$. *input* is expected to be log-probabilities, data type must be float16 or float32 or bfloat16 (only supported by Atlas A2 training series products).
- **target** (Tensor) –For class indices, tensor of shape (), (N) or $(N, d_1, d_2, ..., d_K)$, data type must be int32 or int64. For probabilities, tensor of shape $(N,)$, (N, C) or $(N, C, d_1, d_2, ..., d_K)$, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products).
- **weight** (Tensor, *optional*) –A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$, data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products). Default: `None` .
- **ignore_index** (*int*, *optional*) –Specifies a target value that is ignored and does not contribute to the input gradient. Only valid in class indices, please set it to a negative number in probabilities. Default: `-100` .
- **reduction** (*str*, *optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **label_smoothing** (*float*, *optional*) –Label smoothing values, a regularization tool used to prevent the model from overfitting when calculating Loss. The value range is [0.0, 1.0]. Default: `0.0` .

Returns

Tensor, the data type is the same as *input* .

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint, Tensor
>>> import numpy as np
>>> # Case 1: Indices labels
>>> inputs = Tensor(np.random.randn(3, 5), ms.float32)
>>> target = Tensor(np.array([1, 0, 4]), ms.int32)
>>> output = mint.nn.functional.cross_entropy(inputs, target)
>>> # Case 2: Probability labels
>>> inputs = Tensor(np.random.randn(3, 5), ms.float32)
>>> target = Tensor(np.random.randn(3, 5), ms.float32)
>>> output = mint.nn.functional.cross_entropy(inputs, target)
```

mindspore.mint.nn.functional.binary_cross_entropy

`mindspore.mint.nn.functional.`**`binary_cross_entropy`**(*input*, *target*, *weight=None*, *reduction='mean'*)

Computes the binary cross entropy(Measure the difference information between two probability distributions) between predictive value *input* and target value *target*.

Set *input* as x, *target* as y, output as $\ell(x, y)$, the weight of nth batch of binary cross entropy is w_n. Let,

$$L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

In which, L indicates the loss of all *batch_size*, l indicates the loss of one *batch_size*, and n indicates one *batch_size* in the $1 - N$ range. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Warning:

- The value of *input* must range from *0* to *1*.

Parameters

- **`input`** (`Tensor`) −The predictive value whose data type must be float16 or float32.
- **`target`** (`Tensor`) −The target value which has the same shape and data type as *input*. And the data type is float16 or float32.
- **`weight`** (`Tensor, optional`) −A rescaling weight applied to the loss of each batch element. Its shape must be able to broadcast to that of *input* and *target*. And it must have the same shape and data type as *input*. Default: `None` . If set to `None` , the loss function will not consider any sample weights, and each sample will be treated as having equal importance when calculating the loss.

- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar. Returns Tensor that has the same dtype and shape as *input* if *reduction* is `'none'`. Otherwise, returns a scalar Tensor.

Raises

- **TypeError** –If *input*, *target* or *weight* is not a Tensor.
- **TypeError** –If dtype of *input*, *target* or *weight* (if given) is neither float16 nor float32.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.
- **ValueError** –If shape of *target* is not the same as *input* or *weight* (if given).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0.2, 0.7, 0.1]), mindspore.float32)
>>> target = Tensor(np.array([0., 1., 0.]), mindspore.float32)
>>> weight = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = mint.nn.functional.binary_cross_entropy(input, target, weight)
>>> print(output)
0.38240486
```

mindspore.mint.nn.functional.binary_cross_entropy_with_logits

mindspore.mint.nn.functional.**binary_cross_entropy_with_logits**(*input*, *target*, *weight=None*, *reduction='mean'*, *pos_weight=None*)

Adds sigmoid activation function to *input* as logits, and uses this logits to compute binary cross entropy between the logits and the target. Consistent with the function of *mindspore.ops.binary _cross_entropy_with_logits()* .

Sets input *input* as X, input *target* as Y, input *weight* as W, output as L. Then,

$$
\begin{aligned}
p_{ij} &= sigmoid(X_{ij}) = \frac{1}{1+e^{-X_{ij}}} \\
L_{ij} &= -[Y_{ij}log(p_{ij}) + (1 - Y_{ij})log(1 - p_{ij})]
\end{aligned}
$$

i indicates the i^{th} sample, j indicates the category. Then,

$$
\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}
$$

ℓ indicates the method of calculating the loss. There are three methods: the first method is to provide the loss value directly, the second method is to calculate the average value of all losses, and the third method is to calculate the sum of all losses.

This operator will multiply the output by the corresponding weight. The tensor $weight$ assigns different weights to each piece of data in the batch, and the tensor pos_weight adds corresponding weights to the positive examples of each category.

In addition, it can trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$p_{ij,c} = sigmoid(X_{ij,c}) = \frac{1}{1+e^{-X_{ij,c}}}$$
$$L_{ij,c} = -[P_c Y_{ij,c} * log(p_{ij,c}) + (1 - Y_{ij,c}) log(1 - p_{ij,c})]$$

where c is the class number (c>1 for multi-label binary classification, c=1 for single-label binary classification), n is the number of the sample in the batch and P_c is the weight of the positive answer for the class c. $P_c > 1$ increases the recall, $P_c < 1$ increases the precision.

Parameters

- **input** (Tensor) −Input *input* with shape $(N, *)$ where $*$ means, any number of additional dimensions. The data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **target** (Tensor) −Ground truth label, has the same shape as *input*. The data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **weight** (Tensor, *optional*) −A rescaling weight applied to the loss of each batch element. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None`, *weight* is a Tensor whose value is `1`.
- **reduction** (*str, optional*) −Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **pos_weight** (Tensor, *optional*) −A weight of positive examples. Must be a vector with length equal to the number of classes. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported). Default: `None`, it equals to *pos_weight* is a Tensor whose value is `1`.

Returns

Tensor or Scalar, if *reduction* is `'none'`, it's a tensor with the same shape and type as input *input*. Otherwise, the output is a Scalar.

Raises

- **TypeError** −If input *input*, *target*, *weight*, *pos_weight* is not Tensor.
- **TypeError** −If data type of input *reduction* is not string.
- **ValueError** −If *weight* or *pos_weight* can not be broadcast to a tensor with shape of *input*.
- **ValueError** −If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]), mindspore.float32)
>>> target = Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]), mindspore.float32)
>>> weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> pos_weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> output = mint.nn.functional.binary_cross_entropy_with_logits(input, target, weight, 'mean
↪', pos_weight)
>>> print(output)
0.3463612
```

mindspore.mint.nn.functional.kl_div

mindspore.mint.nn.functional.**kl_div**(*input*, *target*, *reduction='mean'*, *log_target=False*)

Computes the Kullback-Leibler divergence between the *input* and the *target*.

For tensors of the same shape x and y, the updating formulas of KLDivLoss algorithm are as follows,

$$L(x, y) = y \cdot (\log y - x)$$

Then,

$$\ell(x, y) = \begin{cases} L(x, y), & \text{if reduction = 'none';} \\ \text{mean}(L(x, y)), & \text{if reduction = 'mean';} \\ \text{sum}(L(x, y))/x.\text{shape}[0], & \text{if reduction = 'batchmean';} \\ \text{sum}(L(x, y)), & \text{if reduction = 'sum'.} \end{cases}$$

where x represents *input*, y represents *target*, and $\ell(x, y)$ represents the output.

Note: The output aligns with the mathematical definition of Kullback-Leibler divergence only when *reduction* is set to `'batchmean'`.

Parameters

- **input** (Tensor) –The input Tensor. The data type must be float16, float32 or bfloat16(only supported by Atlas A2 training series products).
- **target** (Tensor) –The target Tensor which has the same type as *input*. The shapes of *target* and *input* should be broadcastable.
- **reduction** (*str, optional*) –Specifies the reduction to be applied to the output. Default: `'mean'`.
- **log_target** (*bool, optional*) –Specifies whether *target* is passed in the log space. Default: `False`.

Returns

Tensor, has the same dtype as *input*. If *reduction* is `'none'`, then output has the shape as broadcast result of the *input* and *target*. Otherwise, it is a scalar Tensor.

Raises

- **TypeError** –If neither *input* nor *target* is a Tensor.
- **TypeError** –If dtype of *input* or *target* is not float16, float32 or bfloat16.

- **TypeError** –If dtype of *target* is not the same as *input*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`, `'batchmean'`.
- **ValueError** –If shapes of *target* and *input* can not be broadcastable.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> input = ms.Tensor(np.array([[0.5, 0.5], [0.4, 0.6]]), ms.float32)
>>> target = ms.Tensor(np.array([[0., 1.], [1., 0.]]), ms.float32)
>>> output = mint.nn.functional.kl_div(input, target, reduction='mean', log_target=False)
>>> print(output)
-0.225
```

mindspore.mint.nn.functional.l1_loss

`mindspore.mint.nn.functional.l1_loss`(*input*, *target*, *reduction='mean'*)

Calculate the mean absolute error between the *input* value and the *target* value.

Assuming that the x and y are the predicted value and target value, both are one-dimensional tensors of length N, length N, *reduction* is set to `'none'` , then calculate the loss of x and y without dimensionality reduction.

The formula is as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad \text{with } l_n = |x_n - y_n|,$$

where N is the batch size.

If *reduction* is `'mean'` or `'sum'` , then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Parameters

- **input** (Tensor) –Predicted value, Tensor of any dimension.
- **target** (Tensor) –Target value, usually has the same shape as the *input*. If *input* and *target* have different shapes, make sure they can broadcast to each other.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output. Notice: At least one of the input and target is float type when the reduction is `'mean'` .
  - `'sum'`: the output elements will be summed.

Returns

Tensor or Scalar, if *reduction* is `'none'` , return a Tensor with same shape and dtype as *input*. Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *input* is not a Tensor.
- **TypeError** –If *target* is not a Tensor.
- **ValueError** –If *reduction* is not one of `'none'` , `'mean'` or `'sum'` .

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor([[1, 2, 3], [4, 5, 6]], mstype.float32)
>>> target = Tensor([[6, 5, 4], [3, 2, 1]], mstype.float32)
>>> output = mint.nn.functional.l1_loss(x, target, reduction="mean")
>>> print(output)
3.0
```

mindspore.mint.nn.functional.mse_loss

mindspore.mint.nn.functional.**mse_loss**(*input*, *target*, *reduction='mean'*)

Calculates the mean squared error between the predicted value and the label value.

For detailed information, please refer to *mindspore.nn.MSELoss*.

Parameters

- **input** (Tensor) –Tensor of any dimension. The data type needs to be consistent with the *target*. It should also be broadcastable with the *target*.
- **target** (Tensor) –The input label. Tensor of any dimension. The data type needs to be consistent with the *input*. It should also be broadcastable with the *input*.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'mean'` , `'none'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Returns

- Tensor. If *reduction* is `'mean'` or `'sum'`, the shape of output is *Tensor Scalar*.
- If reduction is `'none'`, the shape of output is the broadcasted shape of **input** and **target** .

Raises

- **ValueError** –If *reduction* is not one of `'mean'` , `'sum'` or `'none'`.
- **ValueError** –If *input* and *target* are not broadcastable.
- **TypeError** –If *input* and *target* are in different data type.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([[1, 1, 1], [1, 2, 2]]), mindspore.float32)
>>> output = mint.nn.functional.mse_loss(logits, labels, reduction='none')
>>> print(output)
[[0. 1. 4.]
 [0. 0. 1.]]
```

mindspore.mint.nn.functional.nll_loss

`mindspore.mint.nn.functional.`**`nll_loss`**(*input*, *target*, *weight=None*, *ignore_index=- 100*, *reduction='mean'*)

Gets the negative log likelihood loss between input and target.

The nll loss with reduction=none can be described as:

$$\ell(x,t) = L = \{l_1,\dots,l_N\}^\top, \quad l_n = -w_{t_n} x_{n,t_n}, \quad w_c = \text{weight}\,[c] \cdot \mathbb{1}\{c \neq \text{ignore\_index}\},$$

where x is the input, t is the target, w is the weight, N is the batch size, c belonging to $[0, C-1]$ is class index, where C is the number of classes.

If *reduction* is not `'None'` (default `'mean'`), then

$$\ell(x,t) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{tn}} l_n, & \text{if reduction} = \text{'mean'}, \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'} \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **`input`** (`Tensor`) – (N) or (N, C) where C = *number of classes* , N = *batch size* , or $(N, C, d_1, d_2, ..., d_K)$ (for high-dimensional data). *input* is expected to be log-probabilities. Data type only supports float32 or float16 or bfloat16(only supported by Atlas A2 training series products).
- **`target`** (`Tensor`) – $()$ or (N) , where the value range is $[0, C-1]$, or $(N, d_1, d_2, ..., d_K)$ for high-dimensional loss, data type must be int32 or int64 or uint8.
- **`weight`** (`Tensor, optional`) – A rescaling weight applied to the loss of each batch element. If not None, the shape is $(C,)$. The data type must be float16 or float32 or bfloat16(only supported by Atlas A2 training series products). It should have the same data type as *input* . Default: `'None'` .
- **`ignore_index`** (`int, optional`) – Specifies a target value that is ignored and does not contribute to the input gradient. Default: `-100` .
- **`reduction`** (`str, optional`) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.

- `'mean'`: compute and return the weighted mean of elements in the output.
- `'sum'`: the output elements will be summed.

Returns

Tensor. The data type is the same as that of *input*.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = mindspore.Tensor(np.random.randn(3, 5), mindspore.float32)
>>> target = mindspore.Tensor(np.array([1, 0, 4]), mindspore.int32)
>>> output = mint.nn.functional.nll_loss(input, target)
```

mindspore.mint.nn.functional.smooth_l1_loss

`mindspore.mint.nn.functional.`**`smooth_l1_loss`**(*input*, *target*, *reduction='mean'*, *beta=1.0*)

Computes smooth L1 loss, a robust L1 loss.

SmoothL1Loss is a Loss similar to MSELoss but less sensitive to outliers as described in the Fast R-CNN by Ross Girshick.

Given two inputs x, y of length N, the SmoothL1Loss can be described as follows:

$$L_i = \begin{cases} \frac{0.5(x_i - y_i)^2}{\text{beta}}, & \text{if } |x_i - y_i| < \text{beta} \\ |x_i - y_i| - 0.5 * \text{beta}, & \text{otherwise.} \end{cases}$$

If *reduction* is not *none*, then:

$$L = \begin{cases} \text{mean}(L_i), & \text{if reduction = 'mean';} \\ \text{sum}(L_i), & \text{if reduction = 'sum'.} \end{cases}$$

Here beta controls the point where the loss function changes from quadratic to linear. beta ≥ 0 , its default value is `1.0` . N is the batch size.

Warning: This is an experimental optimizer API that is subject to change.

Note:

- Arg *input* and *target* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to relatively the highest precision data type.

Parameters

- **`input`** (`Tensor`) –Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions. Supported dtypes:
  - Ascend: float16, float32, bfloat16.

- **target** (Tensor) –Ground truth data, tensor of shape $(N, *)$, same shape as the *input*. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
- **reduction** (*str, optional*) –Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute the mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **beta** (*number, optional*) –A parameter used to control the point where the function will change between L1 to L2 loss. The value should be greater than or equal to zero. Default: `1.0` .

Returns

Tensor, the data type is the same as *input*. If *reduction* is `'none'`, then output is a tensor with the same shape as *input*. Otherwise, the shape of output tensor is ().

Raises

- **TypeError** –If *input* or *target* is not a Tensor.
- **RuntimeError** –If dtype of *input* or *target* is not one of float16, float32, bfloat16.
- **ValueError** –If shape of *input* is not the same as *target*.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.
- **TypeError** –If *beta* is not a float, int or bool.
- **RuntimeError** –If *beta* is less than 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([2, 2, 3]), mindspore.float32)
>>> target = Tensor(np.array([2, 2, 2]), mindspore.float32)
>>> beta = 1.0
>>> reduction_1 = 'none'
>>> output = mint.nn.functional.smooth_l1_loss(input, target, reduction_1, beta)
>>> print(output)
[0.  0.  0.5]
>>> reduction_2 = 'mean'
>>> output = mint.nn.functional.smooth_l1_loss(input, target, reduction_2, beta)
>>> print(output)
0.16666667
>>> reduction_3 = 'sum'
>>> output = mint.nn.functional.smooth_l1_loss(input, target, reduction_3, beta)
>>> print(output)
0.5
```

4.5.9 Vision functions

API Name	Description	Supported Platforms
mindspore.mint.nn.functional.interpolate	Samples the input Tensor to the given size or scale_factor by using one of the interpolate algorithms.	Ascend
mindspore.mint.nn.functional.grid_sample	Given an *input* and a flow-field *grid*, computes the *output* using *input* values and pixel locations from *grid*.	Ascend
mindspore.mint.nn.functional.pad	Pads the input tensor according to the pad.	Ascend
mindspore.mint.nn.functional.pixel_shuffle	Rearrange elements in a tensor according to an upscaling factor.	Ascend

mindspore.mint.nn.functional.interpolate

mindspore.mint.nn.functional.**interpolate**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*)

Samples the input Tensor to the given size or scale_factor by using one of the interpolate algorithms.

Warning: This is an experimental API that is subject to change or deletion.

Note:

- In 'linear' mode, the scenarios, where *scale_factor* is not None and *align_corners* is False, is not supported.
- In 'nearest' mode, there may exist precision problem in the scenarios, where input is 3-D/4-D Tensor and the image is scaled by scale_factor.
- *mode* and *recompute_scale_factor* should be constants.

Parameters

- **input** (Tensor) −Tensor to be resized. Input tensor must be a 3-D, 4-D, or 5-D tensor with shape $(N, C, [optionalD], [optionalH], W)$, with data type of float.
- **size** (*Union[int, tuple[int], list[int]], optional*) −The target size. If size is a tuple or list, its length should be the same as the number of dimensions in input after removing the first two dimensions N, C. One and only one of size and scale_factor can be set to None. Default: None .
- **scale_factor** (*Union[float, tuple[float], list[float]], optional*) −The scale factor of new size of the tensor. If scale_factor is a tuple or list, its length should be the same as the number of dimensions in input after removing the first two dimensions N, C. One and only one of size and scale_factor can be set to None. Default: None .
- **mode** (*str*) −The sampling algorithm. One of 'nearest', 'linear' (3D only), 'bilinear' (4D only), 'trilinear' (5D only), and 'bicubic' (4D only). Default: "nearest" .
- **align_corners** (*bool, optional*) −Whether to use corner alignment for coordinate mapping. Assuming a transformation is applied to the input Tensor along the x-axis, the specific calculation formula is as follows:

```
ori_i = new_length != 1 ? new_i * (ori_length - 1) / (new_length - 1) : 0   # 'align_
↪corners' = True

ori_i = new_length > 1 ? (new_i + 0.5) * ori_length / new_length - 0.5 : 0  # 'align_
↪corners' = False
```

Among them, *ori_length* and *new_length* represent the length of the Tensor before and after transformation along the x-axis respectively; *new_i* represents the coordinate of the i-th element along the x-axis after transformation; *ori_i* represents the corresponding coordinate of the original data along the x-axis.

This is only valid for `'linear'`, `'bilinear'`, or `'bicubic'` modes. Default: `None` .

- **recompute_scale_factor** (*bool, optional*) – Recalculate *scale_factor*. If True, the parameter *size* will be calculated using the value of the *scale_factor*, and finally scaled using the value of *size*. If False, the value of *size* or *scale_factor* will be used for direct interpolation. Default: `None` .

Args Support List and Supported Platforms:

mode	input.dim	align_corners	scale_factor	device
nearest	3	-	√	Ascend
	4	-	√	Ascend
	5	-	√	Ascend
linear	3	√	√	Ascend
bilinear	4	√	×	Ascend
bicubic	4	√	×	Ascend
trilinear	5	√	√	Ascend

- - indicates that there is no such parameter.
- × indicates that this parameter is not currently supported.
- √ indicates that this parameter is supported.

Returns

Tensor, sampled, whose dimensions and dtype are the same as *input*.

Raises

- **TypeError** – *input* is not a Tensor.
- **ValueError** – Both *size* and *scale_factor* are not empty.
- **ValueError** – Both *size* and *scale_factor* are empty.
- **ValueError** – When *size* is a tuple or list, its length is not equal to *input.ndim - 2*.
- **ValueError** – When *scale_factor* is a tuple or list, its length is not equal to *input.ndim - 2*.
- **ValueError** – *mode* is not in the list of supported modes.
- **ValueError** – *input.ndim* is not in the list of supported dimensions for the corresponding mode.
- **ValueError** – *size* is not empty, *recompute_scale_factor* is not empty.
- **ValueError** – *scale_factor* is not in the corresponding list of supported values.
- **ValueError** – *align_corners* is not in the corresponding list of supported values.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, mint
>>> input = Tensor([[[1, 2, 3], [4, 5, 6]]], mindspore.float32)
>>> output = mint.nn.functional.interpolate(input, size=(6,), mode='nearest')
>>> print(output)
[[[1. 1. 2. 2. 3. 3.]
  [4. 4. 5. 5. 6. 6.]]]
```

mindspore.mint.nn.functional.grid_sample

mindspore.mint.nn.functional.**grid_sample**(*input*, *grid*, *mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*)

Given an *input* and a flow-field *grid*, computes the *output* using *input* values and pixel locations from *grid*. Only spatial (4-D) and volumetric (5-D) *input* is supported.

In the spatial (4-D) case, for *input* with shape (N, C, H_{in}, W_{in}) and *grid* with shape $(N, H_{out}, W_{out}, 2)$, the *output* will have shape (N, C, H_{out}, W_{out}).

For each output location *output[n, :, h, w]*, the size-2 vector *grid[n, h, w]* specifies *input* pixel locations *x* and *y*, which are used to interpolate the output value *output[n, :, h, w]*. In the case of 5D inputs, *grid[n, d, h, w]*, specifies the *x*, *y*, *z* pixel locations for interpolating *output[n, :, d, h, w]*. And *mode* argument specifies "nearest" or "bilinear" ("bicubic" is not supported yet) interpolation method to sample the input pixels.

grid specifies the sampling pixel locations normalized by the *input* spatial dimensions. Therefore, it should have most values in the range of $[-1, 1]$.

If *grid* has values outside the range of $[-1, 1]$, the corresponding outputs are handled as defined by *padding_mode*. If *padding_mode* is set to be "zeros", use 0 for out-of-bound grid locations. If *padding_mode* is set to be "border", use border values for out-of-bound grid locations. If *padding_mode* is set to be "reflection", use values at locations reflected by the border for out-of-bound grid locations. For location far away from the border, it will keep being reflected until becoming in bound.

Parameters

- **input** (Tensor) –input with shape of (N, C, H_{in}, W_{in}) (4-D case) or $(N, C, D_{in}, H_{in}, W_{in})$ (5-D case) and dtype of float32 or float64.
- **grid** (Tensor) –flow-field with shape of $(N, H_{out}, W_{out}, 2)$ (4-D case) or $(N, D_{out}, H_{out}, W_{out}, 3)$ (5-D case) and same dtype as *input*.
- **mode** (*str, optional*) –An optional string specifying the interpolation method. The optional values are `'bilinear'`, `'nearest'`. Default: `'bilinear'` . Note: *bicubic* is not supported yet. When *mode="bilinear"* and the input is 5-D, the interpolation mode used internally will actually be trilinear. However, when the input is 4-D, the interpolation mode will legistimately be bilinear. Default: `'bilinear'` .
  - `'nearest'`: Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.
  - `'bilinear'`: Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels, computed using bilinear interpolation. This method produces smoother results compared to nearest neighbor interpolation.
  - `'trilinear'`: Trilinear interpolation. This is an extension of bilinear interpolation to 3D data. It performs bilinear interpolation in the two spatial dimensions and linear interpolation along the third dimension. It is commonly used for volume or 3D image interpolation.

- **padding_mode** (*str, optional*) – An optional string specifying the pad method. The optional values are "zeros", "border" or "reflection". Default: `'zeros'` .
- **align_corners** (*bool, optional*) – If set to *True*, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to *False*, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. Default: `False` .

Returns

Tensor, dtype is the same as *input* and whose shape is (N, C, H_{out}, W_{out}) (4-D) and $(N, C, D_{out}, H_{out}, W_{out})$ (5-D).

Raises

- **TypeError** – If *input* or *grid* is not a Tensor.
- **TypeError** – If the dtypes of *input* and *grid* are inconsistent.
- **TypeError** – If the dtype of *input* or *grid* is not a valid type.
- **TypeError** – If *align_corners* is not a boolean value.
- **ValueError** – If the rank of *input* or *grid* is not equal to 4(4-D case) or 5(5-D case).
- **ValueError** – If the first dimension of *input* is not equal to that of *grid*.
- **ValueError** – If the last dimension of *grid* is not equal to 2(4-D case) or 3(5-D case).
- **ValueError** – If *mode* is not "bilinear", "nearest" or a string value.
- **ValueError** – If *padding_mode* is not "zeros", "border", "reflection" or a string value.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input_x = Tensor(np.arange(16).reshape((2, 2, 2, 2)).astype(np.float32))
>>> grid = Tensor(np.arange(0.2, 1, 0.1).reshape((2, 2, 1, 2)).astype(np.float32))
>>> output = mint.nn.functional.grid_sample(input_x, grid, mode='bilinear', padding_mode=
↪'zeros',
...                                         align_corners=True)
>>> print(output)
[[[[ 1.9       ]
   [ 2.1999998]]
  [[ 5.9       ]
   [ 6.2       ]]]
 [[[10.5       ]
   [10.8       ]]
  [[14.5       ]
   [14.8       ]]]]
```

mindspore.mint.nn.functional.pad

mindspore.mint.nn.functional.**pad**(*input*, *pad*, *mode='constant'*, *value=None*)

Pads the input tensor according to the pad.

> **Warning:** *circular* mode has poor performance and is not recommended.

Parameters

- **input** (Tensor) –Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions.
- **pad** (*Union[tuple[int], list[int], Tensor]*) –Filling position of pad. $\left\lfloor \frac{\text{len(pad)}}{2} \right\rfloor$ dimensions of *input* will be padded.

  Example: to pad only the last dimension of the input tensor, then *pad* has the form (padding_left, padding_right);

  Example: to pad the last 2 dimensions of the input tensor, then use (padding_left, padding_right, padding_top, padding_bottom);

  Example: to pad the last 3 dimensions, use (padding_left, padding_right, padding_top, padding_bottom, padding_front, padding_ba
  and so on.
- **mode** (*str, optional*) –Pad filling mode, `'constant'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'constant'`.

  For `'constant'` mode, please refer to *mindspore.nn.ConstantPad1d* as an example to understand this filling pattern and extend the padding pattern to n dimensions.

  For `'reflect'` mode, please refer to *mindspore.nn.ReflectionPad1d* as an example to understand this filling pattern. The reflect mode is used to pad the last three dimensions of 4D or 5D input, the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.

  For `'replicate'` mode, please refer to *mindspore.nn.ReplicationPad1d* as an example to understand this filling pattern. The replicate mode is used to pad the last three dimensions of 4D or 5D input, the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.

  For `'circular'` mode, the pixels from one edge of the image are wrapped around to the opposite edge, such that the pixel on the right edge of the image is replaced with the pixel on the left edge, and the pixel on the bottom edge is replaced with the pixel on the top edge. The circular mode is used to pad the last three dimensions of 4D or 5D input, the last two dimensions of 3D or 4D input, or the last dimension of 2D or 3D input.
- **value** (*Union[int, float, None], optional*) –Valid only in `'constant'` mode. Set the padding value in `'constant'` mode. If the value is None, 0 is used as the default padding value. Default: `None`.

Returns

Tensor, the tensor after padding.

Raises

- **TypeError** –If *pad* is not an int of tuple or int of list.
- **TypeError** –If *input* is not a Tensor.
- **ValueError** –If length of *pad* is not even.
- **ValueError** –If length of *pad* is greater than 6.
- **ValueError** –If *mode* is not `'constant'` and *value* is neither `None` nor 0.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint
>>> import numpy as np
>>> x = ms.Tensor(np.arange(1 * 2 * 2 * 2).reshape((1, 2, 2, 2)), dtype=ms.float64)
>>> output = mint.nn.functional.pad(x, [1, 0, 0, 1], mode='constant', value=6.0)
>>> print(output)
[[[[6. 0. 1.]
   [6. 2. 3.]
   [6. 6. 6.]]
  [[6. 4. 5.]
   [6. 6. 7.]
   [6. 6. 6.]]]]
```

mindspore.mint.nn.functional.pixel_shuffle

mindspore.mint.nn.functional.**pixel_shuffle**(*input*, *upscale_factor*) → *Tensor*

Rearrange elements in a tensor according to an upscaling factor.

Rearranges elements in a tensor of shape $(*, C \times r^2, H, W)$ to a tensor of shape $(*, C, H \times r, W \times r)$, where r is an upscale factor.

This is useful for implementing efficient sub-pixel convolution with a stride of $1/r$.

For detailed introduction to the pixel_shuffle algorithm, refer to Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network .

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **input** (Tensor) –Tensor of shape $(*, C \times r^2, H, W)$. The dimension of *input* is larger than 2, and the length of third to last dimension can be divisible by the square of *upscale_factor*.
- **upscale_factor** (*int*) –factor to shuffle the input Tensor, and is a positive integer. *upscale_factor* is the above-mentioned r.

Returns

- **output** (Tensor) - Tensor of shape $(*, C, H \times r, W \times r)$.

Raises

- **ValueError** –If *upscale_factor* is not a positive integer.
- **ValueError** –If the length of third to last dimension is not divisible by the square of *upscale_factor*.
- **ValueError** –If the dimension of *input* is less than 3.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import mint
>>> input = mint.randn(1, 9, 4, 4)
>>> output = mint.nn.functional.pixel_shuffle(input, 3)
>>> print(output.shape)
    (1, 1, 12, 12)
```

4.6 mindspore.mint.optim

API Name	Description	Supported Platforms
mindspore.mint.optim.Adam	Implements Adaptive Moment Estimation (Adam) algorithm.	Ascend
mindspore.mint.optim.AdamW	Implements Adam Weight Decay algorithm.	Ascend
mindspore.mint.optim.SGD	Stochastic Gradient Descent optimizer.	Ascend

4.6.1 mindspore.mint.optim.Adam

class mindspore.mint.optim.**Adam**(*params*, *lr=1e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=0.0*, *amsgrad=False*, ***, *maximize=False*)

Implements Adaptive Moment Estimation (Adam) algorithm.

The updating formulas are as follows:

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$
for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$
return θ_t

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups
- **lr** (*Union[int, float, Tensor], optional*) –learning rate. Default: `1e-3`.
- **betas** (*Tuple[float, float], optional*) –The exponential decay rate for the moment estimations. Should be in range (0.0, 1.0). Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) –term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-8`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `0.`.
- **amsgrad** (*bool, optional*) –whether to use the AMSGrad algorithm. Default: `False`.

Keyword Arguments

maximize (`bool, optional`) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the *lr* is not int, float or Tensor.
- **ValueError** –If the *lr* is less than 0.
- **ValueError** –If the *eps* is less than 0.0.
- **ValueError** –If the *betas* is not in the range of [0, 1).
- **ValueError** –If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> from mindspore import mint
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = mint.optim.Adam(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

4.6.2 mindspore.mint.optim.AdamW

class mindspore.mint.optim.**AdamW**(*params*, *lr=1e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=1e-2*, *amsgrad=False*, ***, *maximize=False*)

Implements Adam Weight Decay algorithm.

---

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)

λ(weight decay), *amsgrad*, *maximize*

initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**

if *maximize* :

$g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$

else

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$

$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

$\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$

$\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$

if $amsgrad$

$\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$

$\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$

else

$\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

---

return θ_t

Warning:

- This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .
- For Ascend, it is only supported on platforms above Atlas A2.

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups.
- **lr** (*float, optional*) –learning rate. Default: `1e-3`.
- **betas** (*Tuple[float, float], optional*) –The exponential decay rate for the moment estimations. Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) –term added to the denominator to improve numerical stability. Must be greater than 0. Default: `1e-8`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `1e-2`..
- **amsgrad** (*bool, optional*) –whether to use the AMSGrad algorithm. Default: `False`.

Keyword Arguments

maximize (`bool, optional`) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the learning rate is not float.
- **ValueError** –If the learning rate is less than 0.
- **ValueError** –If the *eps* is less than 0.
- **ValueError** –If the *betas* not in the range of [0, 1).
- **ValueError** –If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> from mindspore.mint import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.AdamW(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
```

(continues on next page)

(continued from previous page)

```
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

4.6.3 mindspore.mint.optim.SGD

class mindspore.mint.optim.**SGD**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*, ***, *maximize=False*)

Stochastic Gradient Descent optimizer.

$$v_{t+1} = u * v_t + gradient * (1 - dampening)$$

If nesterov is True:

$$p_{t+1} = p_t - lr * (gradient + u * v_{t+1})$$

If nesterov is False:

$$p_{t+1} = p_t - lr * v_{t+1}$$

To be noticed, for the first step, $v_{t+1} = gradient$.

Here : p, v and u denote the parameters, accum, and momentum respectively.

Warning: This is an experimental optimizer API, which may be modified or removed in the future. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[bool, int, float, Tensor]*) –learning rate.
- **momentum** (*Union[bool, int, float], optional*) –momentum factor. Default: `0`.
- **weight_decay** (*Union[bool, int, float], optional*) –weight decay (L2 penalty). Must be greater than or equal to 0. Default: `0`..
- **dampening** (*Union[bool, int, float], optional*) –dampening for momentum. Default: `0`.
- **nesterov** (*bool, optional*) –enable Nesterov momentum. If Nesterov is utilized, the momentum must be positive, and the damping must be equal to 0. Default: `False`.

Keyword Arguments

maximize (*bool, optional*) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not bool, int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *momentum* or *weight_decay* value is less than 0.0.
- **ValueError** – If the *momentum*, *dampening* or *weight_decay* value is not bool, int or float.
- **ValueError** – If the *nesterov* and *maximize* are not bool.
- **ValueError** – If the *nesterov* is true, *momentum* is not positive or *dampening* is not 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> from mindspore import mint
>>> from mindspore.mint import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.SGD(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

4.7 mindspore.mint.linalg

4.7.1 Inverses

API Name	Description	Supported Platforms
mindspore.mint.linalg.inv	Compute the inverse of the input matrix.	`Ascend`
mindspore.mint.linalg.matrix_norm	Returns the matrix norm of a given tensor on the specified dimensions.	`Ascend`
mindspore.mint.linalg.norm	Returns the matrix norm or vector norm of a given tensor.	`Ascend`
mindspore.mint.linalg.vector_norm	Returns the vector norm of the given tensor on the specified dimensions.	`Ascend`
mindspore.mint.linalg.qr	Orthogonal decomposition of the input $A = QR$.	`Ascend`

mindspore.mint.linalg.inv

mindspore.mint.linalg.**inv**(*input*)

Compute the inverse of the input matrix.

Parameters

input (Tensor) – A matrix to be calculated. Input *input* must be at least two dimensions, and the size of the last two dimensions must be the same size. And the matrix must be invertible.

Returns

Tensor, has the same type and shape as input *input*.

Raises

- **ValueError** – If the size of the last two dimensions of *input* is not the same.
- **ValueError** – If *input* is not empty and its dimensions are less than 2.
- **ValueError** – If the dimensions of *input* are larger than 6.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import Tensor, mint
>>> from mindspore import dtype as mstype
>>> x = Tensor([[1., 2.], [3., 4.]], mstype.float32)
>>> print(mint.linalg.inv(x))
[[-2.   1. ]
 [ 1.5 -0.5]]
```

mindspore.mint.linalg.matrix_norm

mindspore.mint.linalg.**matrix_norm**(*A*, *ord='fro'*, *dim=(- 2, - 1)*, *keepdim=False*, ***, *dtype=None*)

Returns the matrix norm of a given tensor on the specified dimensions.

ord is the calculation mode of norm. The following norm modes are supported.

ord	norm for matrix
'fro' (Default)	Frobenius norm
'nuc'	nuclear norm
inf	$max(sum(abs(x), dim = 1))$
-inf	$min(sum(abs(x), dim = 1))$
1	$max(sum(abs(x), dim = 0))$
-1	$min(sum(abs(x), dim = 0))$
2	largest singular value
-2	smallest singular value

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **A** (Tensor) –Tensor of shape $(*, m, n)$ where * is zero or more batch dimensions.
- **ord** (*Union[int, inf, -inf, 'fro', 'nuc'], optional*) –norm's mode. refer to the table above for behavior. Default: `'fro'` .
- **dim** (*Tuple(int, int), optional*) –calculate the dimension of the matrix norm. Default: `(-2, -1)` .
- **keepdim** (*bool*) –whether the output Tensor retains the original dimension. Default: `False` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –When set, *A* will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. When *dtype* is `None` , the dtype of *A* is preserved. Default: `None` .

Returns

Tensor, the result of norm calculation on the specified dimension, *dim*.

Raises

- **TypeError** –If *dim* is not a tuple of int.
- **ValueError** –If the length of *dim* is not equal to 2.
- **ValueError** –If *ord* is not in [2, -2, 1, -1, float('inf'), float('-inf'), 'fro', 'nuc'].
- **ValueError** –If two elements of *dim* is same after normalize.
- **ValueError** –If any elements of *dim* is out of range.

Note: Dynamic shape, Dynamic rank and mutable input is not supported in graph mode (mode=mindspore.GRAPH_MODE).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> A = ms.ops.arange(0, 12, dtype=ms.float32).reshape(3, 4)
>>> print(ms.mint.linalg.matrix_norm(A, ord='fro'))
22.494444
>>> print(ms.mint.linalg.matrix_norm(A, ord='nuc'))
24.364643
>>> print(ms.mint.linalg.matrix_norm(A, ord=float('inf')))
38.0
>>> print(ms.mint.linalg.matrix_norm(A, ord=float('-inf')))
6.0
```

mindspore.mint.linalg.norm

mindspore.mint.linalg.**norm**(*A*, *ord=None*, *dim=None*, *keepdim=False*, *, *dtype=None*)

Returns the matrix norm or vector norm of a given tensor.

ord is the calculation mode of norm. The following norm modes are supported.

ord	norm for matrices	norm for vectors
None (default)	Frobenius norm	*2*-norm (see below)
'fro'	Frobenius norm	–not supported –
'nuc'	nuclear norm	–not supported –
inf	$max(sum(abs(x), dim = 1))$	$max(abs(x))$
-inf	$min(sum(abs(x), dim = 1))$	$min(abs(x))$
0	–not supported –	$sum(x! = 0)$
1	$max(sum(abs(x), dim = 0))$	as below
-1	$min(sum(abs(x), dim = 0))$	as below
2	largest singular value	as below
-2	smallest singular value	as below
other *int* or *float*	–not supported –	$sum(abs(x)^{ord})^{(1/ord)}$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **A** (Tensor) –Tensor of shape $(*, n)$ or $(*, m, n)$ where * is zero or more batch dimensions.
- **ord** (*Union[int, float, inf, -inf, 'fro', 'nuc'], optional*) –norm's mode. Refer to the table above for behavior. Default: None .
- **dim** (*Union[int, Tuple(int)], optional*) –calculate the dimension of vector norm or matrix norm. Default: None .
  - When *dim* is int, it will be calculated by vector norm.
  - When *dim* is a 2-tuple, it will be calculated by matrix norm.
  - If *dim* is None and *ord* is None, *A* will be flattened to 1D and the 2-norm of the vector will be calculated.
  - If *dim* is None and *ord* is not None, *A* must be 1D or 2D.
- **keepdim** (*bool*) –whether the output Tensor retains the original dimension. Default: False .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –When set, *A* will be converted to the specified type, *dtype*, before execution, and dtype of returned Tensor will also be *dtype*. Default: None .

Returns

Tensor, the result of norm calculation on the specified dimension, *dim*, has the same dtype as *A*.

Raises

- **ValueError** –If *dim* is out of range.
- **TypeError** –If *dim* is neither an int nor a tuple of int.
- **TypeError** –If *A* is a vector and *ord* is a str.
- **ValueError** –If *A* is a matrices and *ord* is not in valid mode.

- **ValueError** –If two elements of *dim* is same after normalize.
- **ValueError** –If any elements of *dim* is out of range.

Note: Dynamic shape, Dynamic rank and mutable input is not supported in graph mode (mode=mindspore.GRAPH_MODE).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore import mint, ops
>>> data_range = ops.arange(-13, 13, dtype=ms.float32)
>>> x = data_range[data_range != 0]
>>> print(mint.linalg.norm(x))
38.327538
>>> print(mint.linalg.norm(x, 1))
169.0
>>> n = ops.arange(27, dtype=ms.float32).reshape(3, 3, 3)
>>> print(mint.linalg.norm(n, dim=(1, 2)))
[14.282857 39.76179  66.45299 ]
>>> print(mint.linalg.norm(n[0, :, :]))
14.282857
>>> print(mint.linalg.norm(n[1, :, :]))
39.76179
```

mindspore.mint.linalg.vector_norm

mindspore.mint.linalg.**vector_norm**(*x*, *ord=2*, *dim=None*, *keepdim=False*, *, *dtype=None*)

Returns the vector norm of the given tensor on the specified dimensions.

ord is the calculation mode of norm. The following norm modes are supported.

ord	norm for vectors
2 (Default)	2-norm (see below)
inf	$max(abs(x))$
-inf	$min(abs(x))$
0	$sum(x! = 0)$
other int or float	$sum(abs(x)^{ord})^{(1/ord)}$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **x** (Tensor) –Tensor of shape $(*)$ where * is zero s more batch dimensions.
- **ord** (*Union[bool, int, float, inf, -inf], optional*) –norm's mode. refer to the table above for behavior. Default: 2 .

- **dim** (*Union[int, List(int), Tuple(int)], optional*) –The dimensions along which to perform the vector norm calculation. Default: `None` .
  - When *dim* is an integer, a list or a tuple, the norm calculation will be performed across these specified dimensions, while the remaining dimensions will be considered as batch dimensions.
  - When *dim* is None, the norm will be calculated after flattening the Tensor *x* .
- **keepdim** (*bool*) –whether the output Tensor retains the original dimension. Default: `False` .

Keyword Arguments

dtype (*mindspore.dtype*, optional) –When set, *x* will be converted to the specified type, *dtype* before execution, and dtype of returned Tensor will also be *dtype*. When *dtype* is `None` , the dtype of *x* is preserved. Default: `None` .

Returns

Tensor, the result of norm calculation on the specified dimension, *dim*.

Raises

- **TypeError** –If *x* is not a Tensor.
- **TypeError** –If *dim* is neither an int nor a list or tuple.
- **ValueError** –If *ord* is not in [bool, int, float, inf, -inf].
- **ValueError** –The elements of *dim* are duplicate.
- **ValueError** –If any elements of *dim* is out of range.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> x = ms.ops.arange(0, 12, dtype=ms.float32) - 6
>>> print(ms.mint.linalg.vector_norm(x, ord=2))
12.083046
>>> print(ms.mint.linalg.vector_norm(x, ord=float('inf')))
6.0
>>> print(ms.mint.linalg.vector_norm(x, ord=float('-inf')))
0.0
>>> print(ms.mint.linalg.vector_norm(x, ord=0))
11.0
>>> print(ms.mint.linalg.vector_norm(x, ord=4.5))
7.2243643
```

mindspore.mint.linalg.qr

mindspore.mint.linalg.**qr**(*A*, *mode='reduced'*)

Orthogonal decomposition of the input $A = QR$.

Where A is an input tensor, a dimension is at least 2, and A may be represented as a product form of an orthogonal matrix Q and an upper triangular matrix R.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **A** (Tensor) −The calculated matrix, A is at least two-dimensional.
- **mode** (*str, optional*) −Matrix decomposition mode. The options are `reduced`, `complete`, and `r`. The default value is `reduced`.
  - `"reduced"`: For input $A(*, m, n)$ output simplified size $Q(*, m, k)$, $R(*, k, n)$, where k is the minimum value of m and n.
  - `"complete"`: For input $A(*, m, n)$ output full-size $Q(*, m, m)$, $R(*, m, n)$.
  - `"r"`: Only $R(*, k, n)$ in the reduced scenario is calculated, where k is the minimum value of m and n, and Q is returned as an empty tensor.

Returns

- **Q** (Tensor) - The shape is $Q(*, m, k)$ or $(*, m, n)$, has the same dtype as A.
- **R** (Tensor) - The shape is $Q(*, k, n)$ or $(*, m, n)$, has the same dtype as A.

Raises

- **TypeError** −If A is not a Tensor.
- **TypeError** −If the dtype of A is not the float32.
- **ValueError** −If A is not empty and its dimension is less than 2 dimensions.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([[1.0, 1.0, 2.0, 4.0], [1.0, 1.0, 2.0, 4.0]]), mindspore.float32)
>>> output = mindspore.mint.linalg.qr(x)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[-7.07106829e-01, -7.07106769e-01],
[-7.07106769e-01,  7.07106829e-01]]),
Tensor(shape=[2, 4], dtype=Float32, value=
[[-1.41421354e+00, -1.41421354e+00, -2.82842731e+00, -5.65685463e+00],
[ 0.00000000e+00,  3.42285418e-08,  0.00000000e+00,  0.00000000e+00]]))
```

4.8 mindspore.mint.special

4.8.1 Pointwise Operations

API Name	Description	Supported Platforms
mindspore.mint.special.erfc	Compute the complementary error function of input tensor element-wise.	Ascend
mindspore.mint.special.exp2	Calculates the base-2 exponent of the Tensor *input* element by element.	Ascend
mindspore.mint.special.expm1	Compute exponential of the input tensor, then minus 1, element-wise.	Ascend
mindspore.mint.special.log1p	Compute the natural logarithm of (tensor + 1) element-wise.	Ascend
mindspore.mint.special.log_softmax	Applies the Log Softmax function to the input tensor on the specified axis.	Ascend
mindspore.mint.special.round	Returns half to even of a tensor element-wise.	Ascend
mindspore.mint.special.sinc	Compute the normalized sinc of input.	Ascend

mindspore.mint.special.erfc

mindspore.mint.special.**erfc**(*input*)

Compute the complementary error function of input tensor element-wise.

$$\operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> # The datatype of output will be float32 when datatype of input is in [int64,␣
↪bool](Datatype only supported on Ascend).
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.int64)
>>> mindspore.mint.special.erfc(input)
Tensor(shape=[5], dtype=Float32, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
>>>
>>> # Otherwise output has the same dtype as the input.
>>> input = mindspore.tensor([-1, 0, 1, 2, 3], mindspore.float64)
>>> mindspore.mint.special.erfc(input)
Tensor(shape=[5], dtype=Float64, value= [ 1.84270079e+00,  1.00000000e+00,  1.57299207e-01, ␣
↪4.67773498e-03,  2.20904970e-05])
```

mindspore.mint.special.exp2

mindspore.mint.special.**exp2**(*input*)

Calculates the base-2 exponent of the Tensor *input* element by element.

$$out_i = 2^{input_i}$$

Parameters

input (Tensor) –The input Tensor.

Returns

Tensor, which has the same shape as the *input*.

Raises

TypeError –If *input* is not a Tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> x = Tensor(np.array([0.0, 1.0, 2.0, 4.0]), mindspore.float32)
>>> output = mint.special.exp2(x)
>>> print(output)
[ 1. 2. 4. 16.]
```

mindspore.mint.special.expm1

mindspore.mint.special.**expm1**(*input*)

Compute exponential of the input tensor, then minus 1, element-wise.

$$out_i = e^{x_i} - 1$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.0, 1.0, 3.0], mindspore.float32)
>>> output = mindspore.mint.special.expm1(input)
>>> print(output)
[ 0.         1.7182817 19.085537 ]
```

mindspore.mint.special.log1p

mindspore.mint.special.**log1p**(*input*)

Compute the natural logarithm of (tensor + 1) element-wise.

$$out_i = \log_e(input_i + 1)$$

Parameters

input (Tensor) –The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> x = mindspore.tensor([1.0, 2.0, 4.0], mindspore.float32)
>>> output = mindspore.mint.special.log1p(x)
>>> print(output)
[0.6931472 1.0986123 1.609438 ]
```

mindspore.mint.special.log_softmax

mindspore.mint.special.**log_softmax**(*input*, *dim=None*, ***, *dtype=None*)

Applies the Log Softmax function to the input tensor on the specified axis. Supposes a slice in the given axis, x for each element x_i, the Log Softmax function is shown as follows:

$$\text{output}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)}\right),$$

where N is the length of the Tensor.

Parameters

- **input** (Tensor) –The input Tensor.
- **dim** (*int, optional*) –The axis to perform the Log softmax operation. Default: None .

Keyword Arguments

dtype (*mindspore.dtype*, optional) −The desired dtype of returned Tensor. If not set to None, the input Tensor will be cast to *dtype* before the operation is performed. This is useful for preventing overflows. If set to None, stay the same as original Tensor. Default: `None` .

Returns

Tensor, with the same shape as the input.

Raises

- **TypeError** −If *dim* is not an int.
- **ValueError** −If *dim* is not in range [-len(input.shape), len(input.shape)).

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> logits = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> output = mint.special.log_softmax(logits, dim=-1)
>>> print(output)
[-4.4519143 -3.4519143 -2.4519143 -1.4519144 -0.4519144]
```

mindspore.mint.special.round

mindspore.mint.special.**round**(*input*)

Returns half to even of a tensor element-wise.

$$out_i \approx input_i$$

Note: The input data types supported by the Ascend platform include bfloat16 (Atlas training series products are not supported), float16, float32, float64, int32, and int64.

Parameters

input (Tensor) −The input tensor.

Returns

Tensor, has the same shape and type as the *input*.

Raises

TypeError −If *input* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, mint
>>> input = Tensor(np.array([0.8, 1.5, 2.3, 2.5, -4.5]), mindspore.float32)
>>> output = mint.special.round(input)
>>> print(output)
[ 1.  2.  2.  2. -4.]
```

mindspore.mint.special.sinc

mindspore.mint.special.**sinc**(*input*)

Compute the normalized sinc of input.

$$out_i = \begin{cases} \frac{sin(\pi input_i)}{\pi input_i} & input_i \neq 0 \\ 1 & input_i = 0 \end{cases}$$

Parameters

input (Tensor) – The input tensor.

Returns

Tensor

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([0.62, 0.28, 0.43, 0.62], mindspore.float32)
>>> output = mindspore.mint.special.sinc(input)
>>> print(output)
[0.47735003 0.8759357  0.7224278  0.47735003]
```

4.9 mindspore.mint.distributed

API Name	Description	Supported Platforms
mindspore.mint.distributed.all_gather	Gathers tensors from the specified communication group and returns the tensor list which is all gathered.	Ascend
mindspore.mint.distributed.all_gather_into_tensor	Gathers tensors from the specified communication group and returns the tensor which is all gathered.	Ascend
mindspore.mint.distributed.all_gather_object	Aggregates Python objects in a specified communication group.	Ascend
mindspore.mint.distributed.all_reduce	Reduce tensors across all devices in such a way that all deviceswill get the same final result, returns the tensor which is all reduced.	Ascend

continues on next page

Table 32 – continued from previous page

mindspore.mint.distributed.all_to_all	scatter and gather list of tensor to/from all rank according to input/output tensor list.	Ascend
mindspore.mint.distributed.all_to_all_single	scatter and gather input with split size to/from all rank, and return result in a single tensor.	Ascend
mindspore.mint.distributed.barrier	Synchronizes all processes in the specified group.	Ascend
mindspore.mint.distributed.batch_isend_irecv	Batch send and recv tensors asynchronously.	Ascend
mindspore.mint.distributed.broadcast	Broadcasts the tensor to the whole group.	Ascend
mindspore.mint.distributed.broadcast_object_list	Broadcasts the entire group of input Python objects.	Ascend
mindspore.mint.distributed.destroy_process_group	Destroy the user collective communication group.	Ascend
mindspore.mint.distributed.gather	Gathers tensors from the specified communication group.	Ascend
mindspore.mint.distributed.gather_object	Gathers python objects from the whole group in a single process.	Ascend
mindspore.mint.distributed.get_backend	Get the backend of communication process groups.	Ascend
mindspore.mint.distributed.get_global_rank	A function that returns the rank id in the world group corresponding to the rank which id is 'group_rank' in the user group.	Ascend
mindspore.mint.distributed.get_group_rank	Get the rank ID in the specified user communication group corresponding to the rank ID in the world communication group.	Ascend
mindspore.mint.distributed.get_process_group_ranks	Gets the ranks of the specific group and returns the process ranks in the communication group as a list.	Ascend
mindspore.mint.distributed.get_rank	Get the rank ID for the current device in the specified collective communication group.	Ascend
mindspore.mint.distributed.get_world_size	Get the rank size of the specified collective communication group.	Ascend
mindspore.mint.distributed.init_process_group	Init collective communication lib.	Ascend
mindspore.mint.distributed.irecv	Receive tensors from src asynchronously.	Ascend
mindspore.mint.distributed.isend	Send tensors to the specified dest_rank asynchronously.	Ascend
mindspore.mint.distributed.is_available	Checks if distributed module is available.	Ascend
mindspore.mint.distributed.is_initialized	Checks if default process group has been initialized.	Ascend
mindspore.mint.distributed.new_group	Create a new distributed group.	Ascend
mindspore.mint.distributed.P2POp	Object for *batch_isend_irecv* input, to store information of "isend" and "irecv".	Ascend
mindspore.mint.distributed.recv	Receive tensors from src.	Ascend
mindspore.mint.distributed.reduce	Reduces tensors across the processes in the specified communication group, sends the result to the target dst(global rank), and returns the tensor which is sent to the target process.	Ascend

continues on next page

Table 32 – continued from previous page

mindspore.mint.distributed.reduce_scatter	Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.	Ascend
mindspore.mint.distributed.reduce_scatter_tensor	Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.	Ascend
mindspore.mint.distributed.scatter	Scatter tensor evently across the processes in the specified communication group.	Ascend
mindspore.mint.distributed.scatter_object_list	Scatters picklable objects in scatter_object_input_list to the whole group.	Ascend
mindspore.mint.distributed.send	Send tensors to the specified dest_rank.	Ascend

4.9.1 mindspore.mint.distributed.all_gather

mindspore.mint.distributed.**all_gather**(*tensor_list*, *tensor*, *group=None*, *async_op=False*)

Gathers tensors from the specified communication group and returns the tensor list which is all gathered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **tensor_list** (*list[Tensor]*) –Output list.
- **tensor** (Tensor) –The input tensor to be all gathered into tensor.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of input *tensor* is not Tensor, *tensor_list* is not Tensor List, *group* is not a str or async_op is not bool.
- **TypeError** –If size of *tensor_list* is not equal to group size。
- **TypeError** –If the type or shape of *tensor* not equal to the member of *tensor_list*。
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import all_gather
>>> from mindspore import Tensor
>>>
>>> init_process_group()
>>> input_tensor = Tensor(np.ones([2, 8]).astype(np.float32))
>>> out_tensors = [Tensor(np.zeros([2, 8]).astype(np.float32)), Tensor(np.zeros([2, 8]).
 ↪astype(np.float32))]
>>> output = all_gather(out_tensors, input_tensor)
>>> print(out_tensors)
[Tensor(shape=[2, 8], dtype=Float32, value=
[[ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00 ...  1.00000000e+00,  1.00000000e+00,  1.
 ↪00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00 ...  1.00000000e+00,  1.00000000e+00,  1.
 ↪00000000e+00]]),
Tensor(shape=[2, 8], dtype=Float32, value=
[[ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00 ...  1.00000000e+00,  1.00000000e+00,  1.
 ↪00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00 ...  1.00000000e+00,  1.00000000e+00,  1.
 ↪00000000e+00]])]
```

4.9.2 mindspore.mint.distributed.all_gather_into_tensor

mindspore.mint.distributed.**all_gather_into_tensor**(*output_tensor*, *input_tensor*, *group=None*, *async_op=False*)

Gathers tensors from the specified communication group and returns the tensor which is all gathered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **output_tensor** (Tensor) –The output tensor to be all gathered into tensor.If the number of devices in the group is N, then the shape of output tensor is $(N * x_1, x_2, ..., x_R)$.
- **input_tensor** (Tensor) –The input tensor to be all gathered into tensor. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the input_tensor or output_tensor parameter is not Tensor, *group* is not a str, or async_op is not bool.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import mint
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import all_gather_into_tensor
>>> from mindspore import Tensor
>>>
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> input_tensor = Tensor(np.ones([2, 8]).astype(np.float32))
>>> out_tensor = Tensor(np.zeros([4, 8]).astype(np.float32))
>>> output = all_gather_into_tensor(out_tensor, input_tensor)
>>> print(out_tensor)
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

4.9.3 mindspore.mint.distributed.all_gather_object

`mindspore.mint.distributed.`**`all_gather_object`**(*object_list*, *obj*, *group=None*)

Aggregates Python objects in a specified communication group.

Note: Similar to *mindspore.mint.distributed.all_gather()*, but Python objects can be passed in.

Parameters

- **object_list** (*list[Any]*) –Output Python object list.
- **obj** (*Any*) –Python object to be broadcast from current process.

- **group** (*str, optional*) –The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.

Raises

- **TypeError** –*group* is not a str.
- **TypeError** –If size of *object_list* is not equal to group size.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group, get_rank
>>> from mindspore.mint.distributed import all_gather_object
>>> init_process_group()
>>> rank = get_rank()
>>> obj = ["test", {1: 2}]
>>> object_gather_list=[None, None]
>>> all_gather_object(object_gather_list, obj[rank])
>>> print(object_gather_list)
# rank_0
['test', {1: 2}]
# rank_1
['test', {1: 2}]
```

4.9.4 mindspore.mint.distributed.all_reduce

mindspore.mint.distributed.**all_reduce**(*tensor*, *op=ReduceOp.SUM*, *group=None*, *async_op=False*)

Reduce tensors across all devices in such a way that all deviceswill get the same final result, returns the tensor which is all reduced.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **tensor** (Tensor) –The input and output tensor of collective. The shape of tensor is $(x_1, x_2, ..., x_R)$. The function operates in-place.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like sum, prod, max, and min. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.

- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False` .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str, *op* range is illegal or async_op is not bool.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import all_reduce
>>> from mindspore import Tensor
>>>
>>> init_process_group()
>>> tensor = Tensor(np.ones([2, 8]).astype(np.float32))
>>> output = all_reduce(tensor)
>>> print(tensor)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

4.9.5 mindspore.mint.distributed.all_to_all

`mindspore.mint.distributed.`**all_to_all**(*output_tensor_list*, *input_tensor_list*, *group=None*, *async_op=False*)

scatter and gather list of tensor to/from all rank according to input/output tensor list.

Note:

- tensor shape in *output_shape_list* and *input_tensor_list* should be match across ranks.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **output_tensor_list** (*List[Tensor]*) –List of tensors that indicate the gathered from remote ranks.
- **input_tensor_list** (*List[Tensor]*) –List of tensors to scatter to the remote rank.

- **group** (*str, optional*) – The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.
- **async_op** (*bool, optional*) – Whether this operator should be an async operator. Default: `False`.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If not all elements in *input_tensor_list* or *output_tensor_list* are Tensor.
- **TypeError** – If tensors in *input_tensor_list* or *output_tensor_list* are not the same type.
- **TypeError** – If *group* is not str or *async_op* is not bool.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_rank
>>> from mindspore.mint.distributed import all_to_all
>>> from mindspore import Tensor
>>>
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     send_tensor_list = [Tensor(1.), Tensor([[2, 3], [4, 5.]])]
...     recv_tensor_list = [Tensor((0), dtype=ms.float32), Tensor([0, 0.])]
>>> if this_rank == 1:
...     send_tensor_list = [Tensor([2, 2.]), Tensor([4, 5, 6, 7.])]
...     recv_tensor_list = [Tensor([[0, 0.],[0, 0]]), Tensor([0, 0, 0, 0.])]
>>> handle = all_to_all(recv_tensor_list, send_tensor_list)
>>> print(recv_tensor_list)
rank 0:
(Tensor(shape=[], dtype=Float32, value= 1),
 Tensor(shape=[2], dtype=Float32, value= [2.00000000e+00, 2.00000000e+00]))
rank 1:
(Tensor(shape=[2, 2], dtype=Float32, value=
[[2.00000000e+00, 3.00000000e+00],
 [4.00000000e+00, 5.00000000e+00]]),
 Tensor(shape=[4], dtype=Float32, value=[4.00000000e+00, 5.00000000e+00, 6.00000000e+00, 7.
↪00000000e+00]))
```

4.9.6 mindspore.mint.distributed.all_to_all_single

mindspore.mint.distributed.**all_to_all_single**(*output*, *input*, *output_split_sizes=None*, *input_split_sizes=None*, *group=None*, *async_op=False*)

scatter and gather input with split size to/from all rank, and return result in a single tensor.

Note:

- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **output** (Tensor) –the output tensor is gathered concatenated from remote ranks.
- **input** (Tensor) –tensor to be scattered to remote rank.
- **output_split_sizes** (*Union(Tuple(int), List(int)), optional*) –output split size at dim 0. If set to None, it means equally split by world_size. Default: None.
- **input_split_sizes** (*Union(Tuple(int), List(int)), optional*) –input split size at dim 0. If set to None, it means equally split by world_size. Default: None.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group " in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If *input* or *output* is not tensor. *group* is not a str, or async_op is not bool.
- **ValueError** –When *input_split_sizes* is empty, input dim 0 can not be divided by world_size.
- **ValueError** –When *output_split_sizes* is empty, output dim 0 can not be divided by world_size.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore.mint.distributed import init_process_group, get_rank
>>> from mindspore.mint.distributed import all_to_all_single
```

(continues on next page)

(continued from previous page)

```
>>> from mindspore import Tensor
>>> from mindspore.ops import zeros
>>>
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     output = Tensor(np.zeros([3, 3]).astype(np.float32))
...     tensor = Tensor([[0, 1, 2.], [3, 4, 5], [6, 7, 8]])
...     result = all_to_all_single(output, tensor, [2, 1], [2, 1])
...     print(output)
>>> if this_rank == 1:
...     output = Tensor(np.zeros([2, 3]).astype(np.float32))
...     tensor = Tensor([[9, 10., 11], [12, 13, 14]])
...     result = all_to_all_single(output, tensor, [1, 1], [1, 1])
...     print(output)
rank 0:
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 9. 10. 11.]]
rank 1:
[[ 6.  7.  8.]
 [12. 13. 14.]]
```

4.9.7 mindspore.mint.distributed.barrier

mindspore.mint.distributed.**barrier**(*group=None*, *async_op=False*, *device_ids=None*)

Synchronizes all processes in the specified group. Once the process call this operation, it will be blocked until all processes call this operation. After all processes finish calling the operations, the blocked processes will be woken and continue their task.

Parameters

- **group** (*str, optional*) – The communication group to work on. If `None`, which means `"hccl_world_group"` in Ascend. Default: `None`.
- **async_op** (*bool, optional*) – Whether this operator should be an async operator. Default: `False`.
- **device_ids** (*list[int], optional*) – Currently It is a reserved Parameter.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – *group* is not a str or *async_op* is not a bool.
- **RuntimeError** – If backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.communication.comm_func import barrier
>>> # Launch 2 processes.
>>> init_process_group()
>>> barrier()
>>> print("barrier finish!")
barrier finish!
```

4.9.8 mindspore.mint.distributed.batch_isend_irecv

mindspore.mint.distributed.**batch_isend_irecv**(*p2p_op_list*)

Batch send and recv tensors asynchronously.

Note:

- The 'isend' and 'irecv' of *P2POp* in *p2p_op_list* between ranks need to match each other.
- *P2POp* in *p2p_op_list* can only use the same communication group.
- *tag* of *P2POp* in *p2p_op_list* is not support yet.
- *tensor* of *P2POp* in *p2p_op_list* will not be modified by result inplace.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

p2p_op_list (*list[P2POp]*) – list contains *P2POp*. *P2POp* is type of *mindspore.mint.distributed.P2POp*

Returns

list[CommHandle], CommHandle is an async work handle, Currently only one packaging handle is supported.

Raises

- **TypeError** – If *p2p_op_list* is empty or *p2p_op_list* are not all type of *P2POp*.
- **TypeError** – The group name in *p2p_op_list* are not consistent.
- **TypeError** – The *tensor* in *p2p_op_list* are not Tensor.
- **TypeError** – The *op* in *p2p_op_list* are not isend or irecv.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore.mint.distributed import init_process_group, get_rank, get_world_size
>>> from mindspore.mint.distributed import batch_isend_irecv, P2POp
>>> from mindspore import Tensor
>>>
>>> init_process_group()
>>> this_rank = get_rank()
>>> world_size = get_world_size()
>>> next_rank = (this_rank + 1) % world_size
>>> prev_rank = (this_rank + world_size - 1) % world_size
>>>
>>> send_tensor = Tensor(this_rank + 1, dtype=mindspore.float32)
>>> recv_tensor = Tensor(0., dtype=mindspore.float32)
>>>
>>> send_op = P2POp('isend', send_tensor, next_rank)
>>> recv_op = P2POp('irecv', recv_tensor, prev_rank)
>>>
>>> p2p_op_list = [send_op, recv_op]
>>> output = batch_isend_irecv(p2p_op_list)
>>> print(recv_tensor)
rank 0:
2.0
rank 1:
1.0
```

4.9.9 mindspore.mint.distributed.broadcast

mindspore.mint.distributed.**broadcast**(*tensor*, *src*, *group=None*, *async_op=False*)

Broadcasts the tensor to the whole group.

Note:

- The tensors must have the same shape and format in all processes of the collection.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) – Data to be sent if src is the rank of current process, and tensor to be used to save received data otherwise.
- **src** (*int*) – Specifies the rank(global rank) of the process that broadcast the tensor. And only process *src* will broadcast the tensor.

- **group** (*str, optional*) –The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False`.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the *tensor* parameter is not Tensor, *src* is not an integer, *group* is not a string or *async_op* is not bool.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, broadcast
>>> import numpy as np
>>> # Launch 2 processes.
>>>
>>> init_process_group()
>>> data = ms.Tensor(np.arange(8).reshape([2, 4]).astype(np.float32))
>>> handle = broadcast(tensor=data, src=0)
>>> print(data)
[[0. 1. 2. 3.]
 [4. 5. 6. 7.]]
```

4.9.10 mindspore.mint.distributed.broadcast_object_list

mindspore.mint.distributed.**broadcast_object_list**(*object_list*, *src=0*, *group=None*, *device=None*)

Broadcasts the entire group of input Python objects.

Note:

- Similar to *mindspore.mint.distributed.broadcast()*, but Python objects can be passed in.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **object_list** (*list[Any]*) –list of input to be sent if src is the rank of current process, and list to be used to save received data otherwise.
- **src** (*int, optional*) –Specifies the rank(global rank) of the process that broadcast the Python objects. And only process *src* will broadcast the Python objects. Default: `0` .
- **group** (*str, optional*) –The communication group to work on. If `None`, which means `"hccl_world_group "` in Ascend. Default: `None`.
- **device** (*str, optional*) –Currently it is a reserved parameter. Default: `None`.

Raises

- **TypeError** –If *src* is not an integer or *group* is not a string.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group, broadcast_object_list, get_
 ↪rank
>>> init_process_group()
>>> rank = get_rank()
>>> obj = ["test", 12, {1: 2}]
>>> if rank == 1:
...     obj = [None, None, None]
>>> broadcast_object_list(obj)
>>> print(obj)
['test', 12, {1: 2}]
```

4.9.11 mindspore.mint.distributed.destroy_process_group

`mindspore.mint.distributed.`**destroy_process_group**(*group=None*)

Destroy the user collective communication group. If group is None or "hccl_world_group", Destroy all group and release collective communication lib.

Note:

- This method isn't supported in GPU and CPU versions of MindSpore.
- This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

group (*str, optional*) –The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If None, which means "hccl_world_group" in Ascend. Default: None.

Raises

- **TypeError** –If group is not a string.
- **RuntimeError** –If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, destroy_process_group
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> destroy_process_group()
```

4.9.12 mindspore.mint.distributed.gather

mindspore.mint.distributed.**gather**(*tensor*, *gather_list*, *dst=0*, *group=None*, *async_op=False*)

Gathers tensors from the specified communication group. The operation will gather the tensor from processes according to dimension 0.

Note:

- Only the tensor in process *dst* (global rank) will keep the gathered tensor. The other process will keep a tensor list which has no mathematical meaning.
- The tensors must have the same shape and format in all processes of the collection.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The tensor to be gathered.
- **gather_list** (*list[Tensor]*) –List of same-sized tensors to use for gathered data.
- **dst** (*int, optional*) –Specifies the rank(global rank) of the process that receive the tensor. And only process *dst* will receive the gathered tensor. Default: 0 .
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group " in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If the type of input tensor is not Tensor, or gather_list is not Tensor list.
- **TypeError** – If dst is not an integer, group is not a string or async_op is not bool.
- **TypeError** – If size of *gather_list* is not equal to group size.
- **TypeError** – If the type or shape of *tensor* not equal to the member of *gather_list*.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore.mint.distributed import init_process_group, gather
>>> from mindspore import Tensor
>>> # Launch 2 processes.
>>> init_process_group()
>>> input = Tensor(np.arange(4).reshape([2, 2]).astype(np.float32))
>>> outputs = [Tensor(np.zeros([2, 2]).astype(np.float32)),Tensor(np.zeros([2, 2]).astype(np.
↪float32))]
>>> gather(input, outputs, dst=0)
>>> print(outputs)
# rank_0
[Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  1.00000000e+00],
 [ 2.00000000e+00,  3.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  1.00000000e+00], [ 2.00000000e+00,  3.00000000e+00]])]
[Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  1.00000000e+00],
 [ 2.00000000e+00,  3.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  1.00000000e+00], [ 2.00000000e+00,  3.00000000e+00]])]
# rank_1
[Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00],
 [ 0.00000000e+00,  0.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00], [ 0.00000000e+00,  0.00000000e+00]])]
[Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00],
 [ 0.00000000e+00,  0.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 0.00000000e+00,  0.00000000e+00], [ 0.00000000e+00,  0.00000000e+00]])]
```

4.9.13 mindspore.mint.distributed.gather_object

mindspore.mint.distributed.**gather_object**(*obj*, *object_gather_list=None*, *dst=0*, *group=None*)

Gathers python objects from the whole group in a single process.

Note:

- Similar to *mindspore.mint.distributed.gather()*, but Python objects can be passed in.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **obj** (*Any*) –The python objects to be gathered.
- **object_gather_list** (*list[Any], optional*) –List of same-sized tensors to use for gathered data. On the `dst` rank, it should be correctly sized as the size of the group for this collective and will contain the output. Default: `None`.
- **dst** (*int, optional*) –Specifies the rank(global rank) of the process that receive the tensor. And only process *dst* will receive the gathered tensor. Default: `0` .
- **group** (*str, optional*) –The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.

Raises

- **TypeError** –If dst is not an integer, or group is not a string.
- **TypeError** –If size of *object_gather_list* is not equal to group size.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group, gather_object, get_rank
>>> init_process_group()
>>> rank = get_rank()
>>> obj = ["test", {1: 2}]
>>> object_gather_list=[None, None]
>>> gather_object(obj[rank], object_gather_list)
>>> print(object_gather_list)
# rank_0
['test', {1: 2}]
```

4.9.14 mindspore.mint.distributed.get_backend

mindspore.mint.distributed.**get_backend**(*group=None*)

Get the backend of communication process groups.

Note: Only one communication backend is supported by MindSpore for each process. It should be one of *hccl/nccl/mccl*. Currently only support hccl and mccl.

Parameters

group (*str, optional*) – The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*, If None, which means "hccl_world_group" in Ascend. Default: None.

Returns

string, the backend of the group.

Raises

TypeError – If the *group* is not a str.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_backend
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> backend = get_backend()
>>> print("backend is: ", backend)
backend is: hccl
```

4.9.15 mindspore.mint.distributed.get_global_rank

mindspore.mint.distributed.**get_global_rank**(*group*, *group_rank*)

A function that returns the rank id in the world group corresponding to the rank which id is 'group_rank' in the user group.

Note: This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

- **group** (*str*) – The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If None, which means "hccl_world_group" in Ascend.

- **group_rank** (*int*) –Group rank to query.

Returns

An integer scalar with the rank id in the world group.

Raises

- **TypeError** –If the *group* is not a str.
- **TypeError** –If the *group_rank* is not an integer.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 8 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_global_rank, new_group,␣
↪get_rank
>>> ms.set_device(device_target="Ascend")
>>> # Launch 8 processes.
>>> init_process_group()
>>> rank_ids = [0,4]
>>> if get_rank() in rank_ids:
...     group = new_group(rank_ids)
...     world_rank_id = get_global_rank(group, 1)
...     print("world_rank_id is: ", world_rank_id)
#rank 0 and 4:
world_rank_id is: 4
```

4.9.16 mindspore.mint.distributed.get_group_rank

mindspore.mint.distributed.**get_group_rank**(*group*, *global_rank*)

Get the rank ID in the specified user communication group corresponding to the rank ID in the world communication group.

Note: This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

- **group** (*str*) –The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If `None`, which means `"hccl_world_group"` in Ascend.
- **global_rank** (*int*) –A rank ID in the world communication group.

Returns

int, the rank ID in the user communication group.

Raises

- **TypeError** –If global_rank is not an integer or the group is not a string.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 8 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, new_group, get_group_rank,␣
↪get_rank
>>> ms.set_device(device_target="Ascend")
>>> # Launch 8 processes.
>>> init_process_group()
>>> rank_ids = [0,4]
>>> if get_rank() in rank_ids:
...     group = new_group(rank_ids)
...     group_rank_id = get_group_rank(group, 4)
...     print("group_rank_id is: ", group_rank_id)
#rank 0 and 4:
group_rank_id is: 1
```

4.9.17 mindspore.mint.distributed.get_process_group_ranks

mindspore.mint.distributed.**get_process_group_ranks**(*group=None*)

Gets the ranks of the specific group and returns the process ranks in the communication group as a list.

Parameters

group (*str, optional*) –The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If `None`, which means `"hccl_world_group"` in Ascend. Default: `None`.

Returns

List (List[int]), List of process ranks in the specified communication group.

Raises

- **TypeError** –If the *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_process_group_ranks
>>> # Launch 4 processes.
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> output = get_process_group_ranks()
>>> print(output)
[0, 1, 2, 3]
```

4.9.18 mindspore.mint.distributed.get_rank

mindspore.mint.distributed.**get_rank**(*group=None*)

Get the rank ID for the current device in the specified collective communication group.

Note: This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

group (*str, optional*) – The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If None, which means "hccl_world_group" in Ascend. Default: None.

Returns

int, the rank ID of the calling process within the group. return -1, if not part of the group

Raises

TypeError – If group is not a string.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_rank
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> rank_id = get_rank()
>>> print(rank_id)
>>> # the result is the rank_id in world_group
#rank 0: 0
#rank 1: 1
```

4.9.19 mindspore.mint.distributed.get_world_size

mindspore.mint.distributed.**get_world_size**(*group=None*)

Get the rank size of the specified collective communication group.

Note: This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

group (*str, optional*) – The communication group to work on. Normally, the group should be created by *mindspore.mint.distributed.new_group()*. If None, which means "hccl_world_group" in Ascend. Default: None.

Returns

int, the rank size of the group. return -1, if the group is not available.

Raises

TypeError – If group is not a string.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 8 devices.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, get_world_size
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> group_size = get_world_size()
>>> print("group_size is: ", group_size)
group_size is: 8
```

4.9.20 mindspore.mint.distributed.init_process_group

mindspore.mint.distributed.**init_process_group**(*backend='hccl'*, *init_method=None*, *timeout=None*, *world_size=-1*, *rank=-1*, *store=None*, *pg_options=None*, *device_id=None*)

Init collective communication lib. And create a default collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. In Ascend hardware platforms, this API should be set before the definition of any Tensor and Parameter, and the instantiation and execution of any operation and net.

Parameters

- **backend** (*str, optional*) –The backend to ues. default is hccl and now only support hccl.
- **init_method** (*str, invalid*) –URL specifying how to init collective communication group. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.
- **timeout** (*timedelta, invalid*) –Timeout for API executed. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.
- **world_size** (*int, optional*) –Number of the processes participating in the job.
- **rank** (*int, invalid*) –Rank of the current process. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.
- **store** (*Store, invalid*) –Key/Value store accessible to all workers, used to exchange connection/address information. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.
- **pg_options** (*ProcessGroupOptions, invalid*) –process group options specifying what additional options need to be passed in during the construction of specific process group. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.
- **device_id** (*int, invalid*) –the device id to exeute. Provides parameters consistent with pytorch, but is not currently support, setting is invalid.

Raises

- **ValueError** –If *backend* is not hccl.
- **ValueError** –If *world_size* is not equal to -1 or process group number.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails, or the environment variables RANK_ID/MINDSPORE_HCCL_CONFIG_PATH have not been exported when backend is HCCL.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, destroy_process_group
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> destroy_process_group()
```

4.9.21 mindspore.mint.distributed.irecv

mindspore.mint.distributed.**irecv**(*tensor*, *src=0*, *group=None*, *tag=0*)

Receive tensors from src asynchronously.

Note: Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) – Tensor to fill with received data.
- **src** (*int, optional*) – A required integer identifying the source rank(global rank). Default: 0.
- **group** (*str, optional*) – The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **tag** (*int, optional*) – A required integer identifying the send/recv message tag. The message will be received by the Send op with the same "tag". Default: 0. It is a reserved parameter currently.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If the type of *tensor* is not Tensor, If *src* is not an int or *group* is not a str.
- **ValueError** – If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import isend, irecv, get_rank
>>> from mindspore import Tensor
>>> import numpy as np
>>>
# Launch 2 processes, Process 0 sends the array to Process 1.
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...     handle = isend(input_, 1)
...     handle.wait()
>>> if this_rank == 1:
...     x = Tensor(np.zeros([2, 8]).astype(np.float32))
...     handle = irecv(x, src=0)
...     handle.wait()
...     print(x)
rank 1:
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

4.9.22 mindspore.mint.distributed.isend

mindspore.mint.distributed.**isend**(*tensor*, *dst=0*, *group=None*, *tag=0*)

Send tensors to the specified dest_rank asynchronously.

Note: Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) – Tensor to send.
- **dst** (*int, optional*) – A required integer identifying the destination rank(global rank). Default: 0.
- **group** (*str, optional*) – The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **tag** (*int, optional*) – A required integer identifying the send/recv message tag. The message will be received by the Receive op with the same "tag". Default: 0. It is a reserved parameter currently.

Returns

CommHandle, it is an async work handle.

Raises

- **TypeError** – If the *tensor* is not Tensor, *dst* is not an int or *group* is not a str.

- **ValueError** – If the *dst* process rank id is same as the current process.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import isend, irecv, get_rank
>>> from mindspore import Tensor
>>> import numpy as np
>>>
# Launch 2 processes, Process 0 sends the array to Process 1.
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...     handle = isend(input_, 1)
...     handle.wait()
>>> if this_rank == 1:
...     x = Tensor(np.zeros([2, 8]).astype(np.float32))
...     handle = irecv(x, src=0)
...     handle.wait()
...     print(x)
rank 1:
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

4.9.23 mindspore.mint.distributed.is_available

mindspore.mint.distributed.**is_available**()

Checks if distributed module is available.

Note: Always returns *True* because MindSpore always has distributed ability on all platforms.

Returns

bool, whether this distributed module is available.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import is_available
>>> ms.set_device(device_target="Ascend")
>>> is_available()
True
```

4.9.24 mindspore.mint.distributed.is_initialized

mindspore.mint.distributed.**is_initialized**()

Checks if default process group has been initialized.

Returns

bool, whether the default process group has been initialized.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, is_initialized
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> print(is_initialized())
True
```

4.9.25 mindspore.mint.distributed.new_group

mindspore.mint.distributed.**new_group**(*ranks=None*, *timeout=None*, *backend=None*, *pg_options=None*, *use_local_synchronization=False*, *group_desc=None*)

Create a new distributed group.

Note: This method should be used after *mindspore.mint.distributed.init_process_group()*.

Parameters

- **ranks** (*list[int], optional*) –List of ranks of group members. If `None`, will be create the world group. Default is `None`.
- **timeout** (*int, invalid*) –Currently it is a reserved parameter.
- **backend** (*str, invalid*) –Support backend Library, Currently support `"hccl"` and `"mccl"`. when backend is `"hccl"` will use Huawei Collective Communication Library(HCCL). when backend is `"mccl"` will use MindSpore Collective Communication Library(MCCL). If `None`, which means `"hccl"` in Ascend. Default is `None`.
- **pg_options** (*str, invalid*) –Currently it is a reserved parameter.
- **use_local_synchronization** (*bool, invalid*) –Currently it is a reserved parameter.
- **group_desc** (*str, invalid*) –Currently it is a reserved parameter.

Returns

A string with group name. Return "" in the abnormal scenarios.

Raises

TypeError –If list ranks in Group has duplicate rank id.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables. For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.mint.distributed import init_process_group, new_group
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> group = new_group()
>>> print("group is: ", group)
group is: hccl_world_group
```

4.9.26 mindspore.mint.distributed.P2POp

class mindspore.mint.distributed.**P2POp**(*op*, *tensor*, *peer*, *group=None*, *tag=0*)

Object for *batch_isend_irecv* input, to store information of `"isend"` and `"irecv"`.

Note: *tensor* will be modified in-place by final result when *op* is `"irecv"`.

Parameters

- **op** (*Union[str, function]*) –Only string of `"isend"` and `"irecv"` are allowed. Or function of `distributed.isend` and `distributed.irecv` are allowed.

- **tensor** (Tensor) –tensor for sending/receiving.
- **peer** (*int*) –remote global rank for send/receive.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group " in Ascend. Default: None.
- **tag** (*int, optional*) –currently not supported yet. Default: 0.

Returns

P2POp Object.

Raises

- **TypeError** –when *op* is not string or function of 'isend' and 'irecv'.
- **TypeError** –when *tensor* is not type of Tensor or 'peer' is not int.
- **NotImplementedError** –when *tag* is not 0.

Supported Platforms:

Ascend

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore.mint.distributed import P2POp, isend, irecv
>>> from mindspore import Tensor
>>> # Launch 2 processes.
>>> send_tensor = Tensor(1.)
>>> send_op = P2POp('isend', send_tensor, 1)
>>> send_op = P2POp(isend, send_tensor, 1)
>>> recv_tensor = Tensor(0.)
>>> recv_op = P2POp('irecv', recv_tensor, 0)
>>> recv_op = P2POp(irecv, recv_tensor, 0)
```

4.9.27 mindspore.mint.distributed.recv

mindspore.mint.distributed.**recv**(*tensor*, *src=0*, *group=None*, *tag=0*)

Receive tensors from src.

Note: Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –Tensor to fill with received data.
- **src** (*int, optional*) –A required integer identifying the source rank(global rank). Default: 0.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group " in Ascend. Default: None.
- **tag** (*int, optional*) –A required integer identifying the send/recv message tag. The message will be received by the Send op with the same "tag". Default: 0. It is a reserved parameter currently.

Returns

int, If success, return `0`.

Raises

- **TypeError** –If the *tensor* is not Tensor, *src* is not an int or *group* is not a str.
- **ValueError** –If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import send, recv, get_rank
>>> from mindspore import Tensor
>>> import numpy as np
>>>
# Launch 2 processes, Process 0 sends the array to Process 1.
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...     send(input_, 1)
>>> if this_rank == 1:
...     x = Tensor(np.zeros([2, 8]).astype(np.float32))
...     out = recv(x, src=0)
...     print(x)
rank 1:
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

4.9.28 mindspore.mint.distributed.reduce

`mindspore.mint.distributed.`**`reduce`**(*tensor*, *dst*, *op=ReduceOp.SUM*, *group=None*, *async_op=False*)

Reduces tensors across the processes in the specified communication group, sends the result to the target dst(global rank), and returns the tensor which is sent to the target process.

Note:

- Only process with destination rank receives the reduced output.
- Only support PyNative mode, Graph mode is not currently supported.
- Other processes only get a tensor with shape [1], which has no mathematical meaning.

Parameters

- **tensor** (Tensor) –Input and output of the collective. The function operates in-place.
- **dst** (*int*) –The target rank of the process(global rank) that receives the reduced output.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like sum, prod, max, and min. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. If `None`, which means "`hccl_world_group`" in Ascend. Default: `None`.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False`.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to `True`. CommHandle will be None, when *async_op* is `False`.

Raises

- **TypeError** –If the type of *tensor* is not Tensor, any of *op* and *group* is not a str. async_op is not bool or 'op' is invalid.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> from mindspore import mint
>>> import mindspore.nn as nn
>>> from mindspore.mint.distributed import init_process_group, reduce
>>> from mindspore import Tensor
>>> import numpy as np
>>> # Launch 2 processes.
>>> init_process_group()
>>> dest_rank=1
>>> input_tensor = Tensor(np.ones([2, 8]).astype(np.float32))
>>> output = reduce(input_tensor, dest_rank)
>>> print(input_tensor)
Process with rank 0: [[1. 1. 1. 1. 1. 1. 1. 1.]
                      [1. 1. 1. 1. 1. 1. 1. 1.]],
Process with rank 1: [[2. 2. 2. 2. 2. 2. 2. 2.]
                      [2. 2. 2. 2. 2. 2. 2. 2.]],
```

4.9.29 mindspore.mint.distributed.reduce_scatter

mindspore.mint.distributed.**reduce_scatter**(*output*, *input_list*, *op=ReduceOp.SUM*, *group=None*, *async_op=False*)

Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **output** (Tensor) –the output tensor.
- **input_list** (*list[Tensor]*) –List of tensors to reduce and scatter.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like SUM and MAX. Default: ReduceOp.SUM.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of *output* parameter is not Tensor, *input_list* is not Tensor List.
- **TypeError** –If any of *op* and *group* is not a str. async_op is not bool or 'op' is invalid.
- **TypeError** –If size of *input_list* is not equal to group size.
- **TypeError** –If the type or shape of *output* not equal to the member of *input_list*.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore import Tensor
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import reduce_scatter
>>> import numpy as np
>>>
>>> init_process_group()
```

(continues on next page)

(continued from previous page)

```
>>> input_tensors = [Tensor(np.ones([4, 8]).astype(np.float32)), Tensor(np.ones([4, 8]).
↪astype(np.float32))]
>>> output_tensor = Tensor(np.zeros([4, 8]).astype(np.float32))
>>> output = reduce_scatter(output_tensor ,input_tensors)
>>> print(output_tensor)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

4.9.30 mindspore.mint.distributed.reduce_scatter_tensor

mindspore.mint.distributed.**reduce_scatter_tensor**(*output*, *input*, *op=ReduceOp.SUM*, *group=None*, *async_op=False*)

Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **output** (Tensor) –the output tensor has the same dtype as *input_x* with a shape of $(N/rank\_size, *)$
- **input** (Tensor) –The input tensor to be reduced and scattered, suppose it has a shape $(N, *)$, where * means any number of additional dimensions. N must be divisible by rank_size. rank_size refers to the number of cards in the communication group.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like SUM and MAX. Default: ReduceOp.SUM.
- **group** (*str, optional*) –The communication group to work on. If None, which means "hccl_world_group " in Ascend. Default: None.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: False.

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the input and output parameter is not Tensor, any of *op* and *group* is not a str. async_op is not bool or 'op' is invalid.
- **ValueError** –If the first dimension of the input cannot be divided by the rank_size.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import reduce_scatter_tensor
>>> import numpy as np
>>>
>>> ms.set_device(device_target="Ascend")
>>> init_process_group()
>>> input_tensor = Tensor(np.ones([8, 8]).astype(np.float32))
>>> output_tensor = Tensor(np.ones([4, 8]).astype(np.float32))
>>> output = reduce_scatter_tensor(output_tensor ,input_tensor)
>>> print(output_tensor)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

4.9.31 mindspore.mint.distributed.scatter

mindspore.mint.distributed.**scatter**(*tensor*, *scatter_list*, *src=0*, *group=None*, *async_op=False*)

Scatter tensor evently across the processes in the specified communication group.

Note:

- The interface behavior only support Tensor List input and scatter evenly.
- Only the tensor in process *src* (global rank) will do scatter.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) – the output tensor.
- **scatter_list** (*list[Tensor]*) – List of same-sized tensors to scatter. default is None, must be specified on the source rank.
- **src** (*int, optional*) – Specifies the rank(global rank) of the process that send the tensor. And only process *src* will send the tensor.
- **group** (*str, optional*) – The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **async_op** (*bool, optional*) – Whether this operator should be an async operator. Default: False .

Returns

CommHandle, CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of *tensor* parameter is not Tensor, *scatter_list* is not Tensor List.
- **TypeError** –If any of *op* and *group* is not a str. async_op is not bool or 'op' is invalid.
- **TypeError** –If size of *scatter_list* is not equal to group size.
- **TypeError** –If the type or shape of *tensor* not equal to the member of *scatter_list*.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore import Tensor
>>> from mindspore.mint.distributed import init_process_group, scatter
>>> import numpy as np
>>> # Launch 2 processes.
>>>
>>> init_process_group()
>>> inputs = [Tensor(np.ones([2, 2]).astype(np.float32)), Tensor(np.ones([2, 2]).astype(np.
↪float32))]
>>> output = Tensor(np.zeros([2, 2]).astype(np.float32))
>>> scatter(output, inputs, src=0)
>>> print(output)
# rank_0
[[1. 1.]
 [1. 1.]]
# rank_1
[[1. 1.]
 [1. 1.]]
```

4.9.32 mindspore.mint.distributed.scatter_object_list

mindspore.mint.distributed.**scatter_object_list**(*scatter_object_output_list*, *scatter_object_input_list*, *src=0*, *group=None*)

Scatters picklable objects in scatter_object_input_list to the whole group.

Note:

- Similar to *mindspore.mint.distributed.scatter()*, but Python objects can be passed in.

- Only the objects in process *src* (global rank) will do scatter.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **scatter_object_output_list** (*list[Any]*) –Non-empty list whose first element will store the object scattered to this rank.
- **scatter_object_input_list** (*list[Any]*) –List of python objects to scatter. it must be specified on the source rank.
- **src** (*int, optional*) –Specifies the rank(global rank) of the process that send the tensor. And only process *src* will send the tensor. Default: `0` .
- **group** (*str, optional*) –The communication group to work on. If `None`, which means `"hccl_world_group "` in Ascend. Default: `None`.

Raises

- **TypeError** –If *group* is not a str or *src* is not an integer.
- **TypeError** –If size of *scatter_object_input_list* is not equal to group size.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group, scatter_object_list
>>> init_process_group()
>>> obj = ["test",  {1: 2}]
>>> scatter_object_output_list=[None]
>>> scatter_object_list(scatter_object_output_list, obj)
>>> print(scatter_object_output_list)
# rank_0
['test']
# rank_1
[{1: 2}]
```

4.9.33 mindspore.mint.distributed.send

mindspore.mint.distributed.**send**(*tensor*, *dst=0*, *group=None*, *tag=0*)

Send tensors to the specified dest_rank.

Note: Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) – Tensor to send.
- **dst** (*int, optional*) – A required integer identifying the destination rank(global rank). Default: 0.
- **group** (*str, optional*) – The communication group to work on. If None, which means "hccl_world_group" in Ascend. Default: None.
- **tag** (*int, optional*) – A required integer identifying the send/recv message tag. The message will be received by the Receive op with the same "tag". Default: 0. It is a reserved parameter currently.

Raises

- **TypeError** – If the *tensor* is not Tensor, *dst* is not an int or *group* is not a str.
- **ValueError** – If the *dst* process rank id is same as the current process.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore.mint.distributed import init_process_group
>>> from mindspore.mint.distributed import send, recv, get_rank
>>> from mindspore import Tensor
>>> import numpy as np
>>>
# Launch 2 processes, Process 0 sends the array to Process 1.
>>> init_process_group()
>>> this_rank = get_rank()
>>> if this_rank == 0:
...     input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...     send(input_, 1)
>>> if this_rank == 1:
...     x = Tensor(np.zeros([2, 8]).astype(np.float32))
...     out = recv(x, src=0)
...     print(x)
rank 1:
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

CHAPTER

FIVE

MINDSPORE.COMMON.INITIALIZER

Initializer for cell parameters.

class mindspore.common.initializer.**Constant**(*value*)

Generates an array with constant value in order to initialize a tensor.

Parameters

value (*Union[int, numpy.ndarray]*) – The value to initialize.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Constant
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Constant(3), [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**Dirac**(*groups=1*)

Generates an array with the Dirac delta function in order to initialize a tensor. It's usually used in convolution layers, preserves as many identities of the inputs as possible.

Parameters

groups (*int*) – The number of groups in convolution layer. Each group applies the same initialization. Default: `1` .

Raises

- **ValueError** – If the dimension of the initialized tensor is not in [3, 4, 5].
- **ValueError** – The first dimension of the initialized tensor cannot be divisible by group.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Dirac
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Dirac(groups=2), [6, 4, 3, 3], mindspore.float32))
>>> w2 = Parameter(initializer("dirac", [6, 4, 3, 3], mindspore.float32))
```

class mindspore.common.initializer.**HeNormal**(*negative_slope=0*, *mode='fan_in'*, *nonlinearity='leaky_relu'*)

Generates an array with values sampled from HeKaiming Normal distribution $N(0, \text{sigma}^2)$ in order to initialize a tensor, where

$$sigma = \frac{gain}{\sqrt{fan\_mode}}$$

where *gain* is an optional scaling factor. fan_mode is the number of input or output units of the weight tensor, depending on the *mode* is 'fan_in' or 'fan_out'.

For details of HeNormal algorithm, please check https://arxiv.org/abs/1502.01852.

Parameters

- **negative_slope** (*int, float*) –The negative slope of the rectifier used after this layer (only used when *nonlinearity* is 'leaky_relu'). Default: `0` .
- **mode** (*str*) –Either `'fan_in'` or `'fan_out'` . Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass. Default: `'fan_in'` .
- **nonlinearity** (*str*) –The non-linear function, recommended to use only with `'relu'` or `'leaky_relu'` . Default: `'leaky_relu'` .

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, HeNormal
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(HeNormal(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('he_normal', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**HeUniform**(*negative_slope=0*, *mode='fan_in'*, *nonlinearity='leaky_relu'*)

Generates an array with values sampled from HeKaiming Uniform distribution $U(-\text{boundary}, \text{boundary})$ in order to initialize a tensor, where

$$boundary = \text{gain} \times \sqrt{\frac{3}{fan\_mode}}$$

where *gain* is an optional scaling factor. If *mode* is `'fan_in'`, fan_mode in the formula is the number of input units of the weight tensor. If *mode* is `'fan_out'`, fan_mode is the number of output units of the weight tensor.

For details of HeUniform algorithm, please check https://arxiv.org/abs/1502.01852.

Parameters

- **negative_slope** (*int, float, bool*) –The negative slope of the rectifier used after this layer (only used when *nonlinearity* is 'leaky_relu'). Default: `0` .
- **mode** (*str*) –Either `'fan_in'` or `'fan_out'` . Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass. Default: `'fan_in'` .
- **nonlinearity** (*str*) –The non-linear function, recommended to use only with `'relu'` or `'leaky_relu'` . Default: `'leaky_relu'` .

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, HeUniform
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(HeUniform(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('he_uniform', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**Identity**(***kwargs*)

Generates a 2 dimension identity matrix array in order to initialize a tensor.

Raises

ValueError – If the dimension of input tensor is not equal to 2.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Identity
>>> from mindspore import Parameter
>>> w1 = initializer(Identity(), [2, 3], mindspore.float32)
>>> w2 = initializer('identity', [2, 3], mindspore.float32)
```

class mindspore.common.initializer.**Initializer**(***kwargs*)

The abstract base class of the initializer.

Note: Initializers are intended to be used for delayed initialization in parallel mode rather than Tensor initialization. If you have to use Initializers to create a Tensor, *mindspore.Tensor.init_data()* should be followed in most of the cases. For more information, please refer to *mindspore.Tensor.init_data()* .

Parameters

kwargs (*dict*) – Keyword arguments for Initializer.

class mindspore.common.initializer.**Normal**(*sigma=0.01*, *mean=0.0*)

Generates an array with values sampled from Normal distribution $N(\text{sigma}, \text{mean})$ in order to initialize a tensor.

$$f(x) = \frac{1}{\sqrt{2 * } * sigma} exp(-\frac{(x - mean)^2}{2 * sigma^2})$$

Parameters

- **sigma** (*float*) – The standard deviation of Normal distribution. Default: `0.01` .
- **mean** (*float*) – The mean of Normal distribution. Default: `0.0` .

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Normal
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Normal(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('normal', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**One**(***kwargs*)

Generates an array with constant value of one in order to initialize a tensor.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, One
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(One(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('ones', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**Orthogonal**(*gain=1.*)

Generates a (semi) orthogonal matrix array in order to initialize a tensor. The dimension of input tensor must have at least 2 dimensions. If the dimension is greater than 2, the trailing dimensions will be flattened.

Parameters

gain (*float*) – An optional scaling factor. Default: `1.0`.

Raises

ValueError – If the dimension of input tensor is less than 2.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Orthogonal
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Orthogonal(gain=2.), [2, 3, 4], mindspore.float32))
>>> w2 = Parameter(initializer('orthogonal', [2, 3, 4], mindspore.float32))
```

class mindspore.common.initializer.**Sparse**(*sparsity*, *sigma=0.01*)

Generates a 2 dimension sparse matrix array in order to initialize a tensor. The non-zero positions will be filled with the value sampled from the normal distribution $N(0, sigma)$.

Parameters

- **sparsity** (*float*) – The fraction of elements being set to zero in each column.
- **sigma** (*float*) – The standard deviation of the normal distribution. Default: `0.01`.

Raises

ValueError – If the dimension of input tensor is not equal to 2.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Sparse
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Sparse(sparsity=0.1, sigma=0.01), [5, 8], mindspore.float32))
```

class mindspore.common.initializer.**TruncatedNormal**(*sigma=0.01*, *mean=0.0*, *a=- 2.0*, *b=2.0*)

Generates an array with values sampled from Truncated Normal distribution in order to initialize a tensor.

Parameters

- **sigma** (*float*) – The standard deviation of Truncated Normal distribution. Default: `0.01`.
- **mean** (*float*) – The mean of Truncated Normal distribution. Default: `0.0`.
- **a** (*float*) – The lower bound of the truncated interval. Default: `-2.0`.
- **b** (*float*) – The upper bound of the truncated interval. Default: `2.0`.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, TruncatedNormal
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(TruncatedNormal(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('truncatedNormal', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**Uniform**(*scale=0.07*)

Generates an array with values sampled from Uniform distribution $U(-\text{scale}, \text{scale})$ in order to initialize a tensor.

Parameters

scale (*float*) – The bound of the Uniform distribution. Default: `0.07`.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Uniform
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Uniform(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('uniform', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**VarianceScaling**(*scale=1.0*, *mode='fan_in'*, *distribution='truncated_normal'*)

Generates an random array with scaling in order to initialize a tensor.

When *distribution* is 'truncated_normal' or 'untruncated_normal', the value will be sampled from truncated or untruncated normal distribution with a mean of 0 and a scaled standard deviation $stddev = \sqrt{\frac{scale}{n}}$.

n will be the number of input units if *mode* is `'fan_in'`, while n will be the number of output units if *mode* is `'fan_out'`. n will be the average of `'fan_in'` and `'fan_out'` if *mode* is `'fan_avg'`.

When *distribution* is `'uniform'`, the value will be sampled from a uniform distribution within the limit of $[-\sqrt{\frac{3*scale}{n}}, \sqrt{\frac{3*scale}{n}}]$.

Parameters

- **scale** (*float*) –The scaling factor. Default: `1.0`.
- **mode** (*str*) –Should be `'fan_in'`, `'fan_out'` or `'fan_avg'`. Default: `'fan_in'`.
- **distribution** (*str*) –The type of distribution chose to sample values. It should be `'uniform'`, `'truncated_normal'` or `'untruncated_normal'`. Default: `'truncated_normal'`.

Raises

- **ValueError** –If *scale* is not greater than 0.
- **ValueError** –If *mode* is not `'fan_in'`, `'fan_out'` or `'fan_avg'`.
- **ValueError** –If *distribution* is not `'uniform'`, `'truncated_normal'` or `'untruncated_normal'`.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, VarianceScaling
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(VarianceScaling(scale=1.0, mode='fan_out',
...                                            distribution='untruncated_normal'), [2, 3],␣
↪mindspore.float32))
>>> w2 = Parameter(initializer('varianceScaling', [2, 3], mindspore.float32))
```

class mindspore.common.initializer.**XavierNormal**(*gain=1*)

Generates an array with values sampled from Xavier normal distribution $N(0, \text{sigma}^2)$ in order to initialize a tensor, where

$$sigma = gain * \sqrt{\frac{2}{n_{in} + n_{out}}}$$

where $gain$ is an optional scaling factor, n_{in} is the number of input units in the weight tensor, n_{out} is the number of output units in the weight tensor.

Parameters

gain (*float*) –An optional scaling factor. Default: `1`.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, XavierNormal
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(XavierNormal(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('xavier_normal', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**XavierUniform**(*gain=1*)

Generates an array with values sampled from Xavier uniform distribution $U(-\text{boundary}, \text{boundary})$ in order to initialize a tensor, where

$$boundary = gain * \sqrt{\frac{6}{n_{in} + n_{out}}}$$

where $gain$ is an optional scaling factor. n_{in} is the number of input units in the weight tensor, n_{out} is the number of output units in the weight tensor.

For details of XavierUniform algorithm, please check http://proceedings.mlr.press/v9/glorot10a.html.

Parameters

gain (*float*) –An optional scaling factor. Default: `1`.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, XavierUniform
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(XavierUniform(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('xavier_uniform', [1, 2, 3], mindspore.float32))
```

class mindspore.common.initializer.**Zero**(***kwargs*)

Generates an array with constant value of zero in order to initialize a tensor.

Examples

```
>>> import mindspore
>>> from mindspore.common.initializer import initializer, Zero
>>> from mindspore import Parameter
>>> w1 = Parameter(initializer(Zero(), [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('zeros', [1, 2, 3], mindspore.float32))
```

mindspore.common.initializer.**initializer**(*init*, *shape=None*, *dtype=mstype.float32*)

Create and initialize a tensor.

Parameters

- **init** (*Union[Tensor, str, Initializer, numbers.Number]*) – Initialize value.
  - *Tensor*: The tensor will be called to initialize tensor.
  - *str*: The *init* should be the alias of the class inheriting from *Initializer* and the corresponding class will be called in practice. The value of *init* can be `"normal"`, `"ones"` or `"zeros"`, etc.
  - *Initializer*: The *init* should be the class inheriting from *Initializer* to initialize tensor.
  - *numbers.Number*: The *Constant* will be called to initialize tensor.
- **shape** (*Union[tuple, list, int]*) – The shape of the initialized tensor. Default: `None` .
- **dtype** (*mindspore.dtype*) – The type of data in initialized tensor. Default: `mstype.float32` .

Returns

Returns a Tensor with the shape specified by the input *shape*. If *shape* is `None`, the returned Tensor will have the same shape as `init`.

Raises

- **TypeError** – The type of the argument 'init' is not correct.
- **ValueError** – The shape of the tensor which is passed through 'init' is not the same as that passed by 'shape'.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.common.initializer import initializer, One
>>> from mindspore import Parameter
>>> data = Tensor(np.zeros([1, 2, 3]), mindspore.float32)
>>> w1 = Parameter(initializer(data, [1, 2, 3], mindspore.float32))
>>> w2 = Parameter(initializer('ones', [1, 2, 3], mindspore.float32))
>>> w3 = Parameter(initializer(One(), [1, 2, 3], mindspore.float32))
>>> w4 = Parameter(initializer(0, [1, 2, 3], mindspore.float32))
```

CHAPTER

SIX

MINDSPORE.AMP

6.1 Loss Scale

mindspore.amp.LossScaler	Loss scaler abstract class when using mixed precision.
mindspore.amp.DynamicLossScaler	Manager for dynamically adjusting the loss scaling factor.
mindspore.amp.StaticLossScaler	Static Loss scale class.
mindspore.amp.LossScaleManager	Loss scale (Magnification factor of gradients when mix precision is used) manager abstract class when using mixed precision.
mindspore.amp.DynamicLossScaleManager	Loss scale(Magnification factor of gradients when mix precision is used) manager with loss scale dynamically adjusted, inherits from *mindspore.amp.LossScaleManager*.
mindspore.amp.FixedLossScaleManager	Loss scale (Magnification factor of gradients when mix precision is used) manager with a fixed loss scale value, inherits from *mindspore.amp.LossScaleManager*.

6.1.1 mindspore.amp.LossScaler

class mindspore.amp.**LossScaler**

Loss scaler abstract class when using mixed precision.

Derived class needs to implement all of its methods. During training, *scale* and *unscale* is used to scale and unscale the loss value and gradients to avoid overflow, *adjust* is used to update the loss scale value.

For more information, refer to the tutorials.

> **Warning:** This is an experimental API that is subject to change or deletion.

Examples

```
>>> from mindspore.amp import LossScaler, _grad_scale_map, _grad_unscale_map
>>> from mindspore import ops, Parameter, Tensor
>>> from mindspore.common import dtype as mstype
>>>
>>> class MyLossScaler(LossScaler):
...     def __init__(self, scale_value):
...         self.scale_value = Parameter(Tensor(scale_value, dtype=mstype.float32), name=
↪"scale_value")
```

(continues on next page)

(continued from previous page)

```
...
...     def scale(self, inputs):
...         inputs = mutable(inputs)
...         return _grad_scale_map(self.scale_value, inputs)
...
...     def unscale(self, inputs):
...         inputs = mutable(inputs)
...         return _grad_unscale_map(self.scale_value, inputs)
...
...     def adjust(self, grads_finite):
...         scale_mul_factor = self.scale_value * self.scale_factor
...         scale_value = ops.select(grads_finite, scale_mul_factor, self.scale_value)
...         ops.assign(self.scale_value, scale_value)
...         return True
>>>
>>> loss_scaler = MyLossScaler(1024)
```

abstract adjust(*grads_finite*)

Adjust the *scale_value* dependent on whether grads are finite.

Parameters

grads_finite (Tensor) – a scalar bool Tensor indicating whether the grads are finite.

abstract scale(*inputs*)

Scaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

abstract unscale(*inputs*)

Unscaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

6.1.2 mindspore.amp.DynamicLossScaler

class mindspore.amp.**DynamicLossScaler**(*scale_value*, *scale_factor*, *scale_window*)

Manager for dynamically adjusting the loss scaling factor.

Dynamic loss scaling tries to determine the largest loss scale value that will keep gradients finite. It does this by increasing the loss scale every *scale_window* steps by *factor* if the grads remain finite, otherwise it reduces the loss scale by *1 / factor* and resets the counter.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **scale_value** (*Union(float, int)*) – The initial loss scale value.
- **scale_factor** (*int*) – The scale factor.
- **scale_window** (*int*) – Maximum continuous training steps that do not have overflow to increase the loss scale.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import amp, Tensor
>>> import numpy as np
>>> loss_scaler = amp.DynamicLossScaler(scale_value=2**10, scale_factor=2, scale_window=1)
>>> grads = (Tensor(np.array([np.log(-1), 1.0]), mindspore.float16),
...              Tensor(np.array([0.2]), mindspore.float16))
>>> unscaled_grads = loss_scaler.unscale(grads)
>>> grads_finite = amp.all_finite(unscaled_grads)
>>> loss_scaler.adjust(grads_finite)
True
>>> print(loss_scaler.scale_value.asnumpy())
512.0
```

adjust(*grads_finite*)

Adjust the *scale_value* dependent on whether grads are finite.

Parameters

grads_finite (Tensor) – a scalar bool Tensor indicating whether the grads are finite.

Tutorial Examples:

- Automatic Mix Precision - Loss Scaling

scale(*inputs*)

Scaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

Returns

Union(Tensor, tuple(Tensor)), the scaled value.

Tutorial Examples:

- Automatic Mix Precision - Loss Scaling

unscale(*inputs*)

Unscaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

Returns

Union(Tensor, tuple(Tensor)), the unscaled value.

Tutorial Examples:

- Automatic Mix Precision - Loss Scaling

6.1.3 mindspore.amp.StaticLossScaler

class mindspore.amp.**StaticLossScaler**(*scale_value*)

Static Loss scale class.

Scales and unscales loss or gradients by a fixed constant.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

scale_value (*Union(float, int)*) – The initial loss scale value.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import amp, Tensor
>>> import numpy as np
>>> loss_scaler = amp.StaticLossScaler(scale_value=2**10)
>>> loss_value = Tensor([1.], mindspore.float32)
>>> scaled_loss_value = loss_scaler.scale(loss_value)
>>> print(scaled_loss_value)
[1024.]
>>> grads = (Tensor(np.array([1.5, 1.0]), mindspore.float16),
...          Tensor(np.array([1.2]), mindspore.float16))
>>> unscaled_grads = loss_scaler.unscale(grads)
>>> print(unscaled_grads)
(Tensor(shape=[2], dtype=Float16, value= [ 1.4648e-03,  9.7656e-04]),
Tensor(shape=[1], dtype=Float16, value= [ 1.1721e-03]))
```

adjust(*grads_finite*)

Adjust *scale_value* in *LossScaler*. *scale_value* is fixed in *StaticLossScaler*, so this method return False directly.

Parameters

grads_finite (Tensor) – a scalar bool Tensor indicating whether the grads are finite.

scale(*inputs*)

Scaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

Returns

Union(Tensor, tuple(Tensor)), the scaled value.

unscale(*inputs*)

Unscaling inputs by *scale_value*.

Parameters

inputs (*Union(Tensor, tuple(Tensor))*) – the input loss value or gradients.

Returns

Union(Tensor, tuple(Tensor)), the unscaled value.

6.1.4 mindspore.amp.LossScaleManager

class mindspore.amp.**LossScaleManager**

Loss scale (Magnification factor of gradients when mix precision is used) manager abstract class when using mixed precision.

Derived class needs to implement all of its methods. *get_loss_scale* is used to get current loss scale value. *update_loss_scale* is used to update loss scale value, *update_loss_scale* will be called during the training. *get_update_cell* is used to get the instance of *mindspore.nn.Cell* that is used to update the loss scale, the instance will be called during the training. Currently, the *get_update_cell* is mostly used.

For example, *mindspore.amp.FixedLossScaleManager* and *mindspore.amp.DynamicLossScaleManager*.

get_loss_scale()

Get the value of loss scale, which is the amplification factor of the gradients.

get_update_cell()

Get the instance of *mindspore.nn.Cell* that is used to update the loss scale.

update_loss_scale(*overflow*)

Update the loss scale value according to the status of *overflow*.

Parameters

overflow (*bool*) – Whether the overflow occurs during the training.

6.1.5 mindspore.amp.DynamicLossScaleManager

class mindspore.amp.**DynamicLossScaleManager**(*init_loss_scale=2 ** 24*, *scale_factor=2*, *scale_window=2000*)

Loss scale(Magnification factor of gradients when mix precision is used) manager with loss scale dynamically adjusted, inherits from *mindspore.amp.LossScaleManager*.

Parameters

- **init_loss_scale** (*float, optional*) – Initialize loss scale. Default: `2 ** 24`.
- **scale_factor** (*int, optional*) – Coefficient of increase and decrease. Default: `2`.
- **scale_window** (*int, optional*) – Maximum continuous normal steps when there is no overflow. Default: `2000`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import amp, nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_scale_manager = amp.DynamicLossScaleManager()
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = ms.Model(net, loss_scale_manager=loss_scale_manager, optimizer=optim)
```

get_drop_overflow_update()
Whether to drop optimizer update for current step when there is an overflow.

Returns

bool, always True.

get_loss_scale()
Get the current loss scale value.

Returns

float, *loss_scale* value.

get_update_cell()
Returns the instance of *mindspore.nn.Cell* that is used to update the loss scale which will be called at *mindspore.nn.TrainOneStepWithLossScaleCell*.

Returns

mindspore.nn.DynamicLossScaleUpdateCell.

update_loss_scale(*overflow*)
Update the loss scale value according to the status of *overflow*. If overflow occurs, decrease loss scale per *scale_window*, otherwise, increase the loss scale.

Parameters

overflow (*bool*) –Whether it overflows.

6.1.6 mindspore.amp.FixedLossScaleManager

class mindspore.amp.**FixedLossScaleManager**(*loss_scale=128.0*, *drop_overflow_update=True*)
Loss scale (Magnification factor of gradients when mix precision is used) manager with a fixed loss scale value, inherits from *mindspore.amp.LossScaleManager*.

Parameters

- **loss_scale** (*float, optional*) –Magnification factor of gradients. Note that if *drop_overflow_update* is set to `False` , the value of *loss_scale* in optimizer should be set to the same as here. Default: `128.0` .
- **drop_overflow_update** (*bool, optional*) –Whether to execute optimizer if there is an overflow. If `True` , the optimizer will not executed when overflow occurs. Default: `True` .

Examples

```
>>> import mindspore as ms
>>> from mindspore import amp, nn
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_scale = 1024.0
>>> loss_scale_manager = amp.FixedLossScaleManager(loss_scale, False)
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9, loss_
↪scale=loss_scale)
>>> model = ms.Model(net, loss_scale_manager=loss_scale_manager, optimizer=optim)
```

get_drop_overflow_update()
Get *drop_overflow_update*, whether to drop optimizer update for current step when there is an overflow.

Returns

bool, *drop_overflow_update* value.

get_loss_scale()
Get loss scale value.

Returns

bool, *loss_scale* value.

get_update_cell()
Returns the instance of *mindspore.nn.Cell* that used to update the loss scale which will be called at *mindspore.nn.TrainOneStepWithLossScaleCell*. As the loss scale is fixed in this class, the instance will do nothing.

Returns

None or *mindspore.nn.FixedLossScaleUpdateCell*. Instance of *mindspore.nn.FixedLossScaleUpdateCell* when *drop_overflow_update* is `True`. None when *drop_overflow_update* is `False`.

update_loss_scale(*overflow*)
Update loss scale value. The interface at *mindspore.amp.FixedLossScaleManager* will do nothing.

Parameters

overflow (*bool*) –Whether it overflows.

6.2 Dtype Autocast

mindspore.amp.auto_mixed_precision	Returns a network processed with auto mixed precision.
mindspore.amp.build_train_network	Build the mixed precision training cell automatically.
mindspore.amp.custom_mixed_precision	When the *white_list* is provided, primitives and cells in *white_list* will perform the precision conversion.
mindspore.amp.get_black_list	Provide a copy of internal black list used by auto mixed precision with *amp_level* set to `O2`.
mindspore.amp.get_white_list	Provide a copy of internal white list used by auto mixed precision with *amp_level* set to `O1`.

6.2.1 mindspore.amp.auto_mixed_precision

mindspore.amp.**auto_mixed_precision**(*network*, *amp_level='O0'*, *dtype=mstype.float16*)

Returns a network processed with auto mixed precision.

This interface will automatically perform mixed-precision processing on the input network, and the cells and operators in the processed network will add precision conversion operations to calculate with lower precision: `mstype.float16` or `mstype.bfloat16` . Inputs and parameters of cells and operators are converted to lower precision float, and calculation results are converted back to full precision float, i.e. `mstype.float32` .

The *amp_level* and its corresponding lists determine which cells and operators are converted.

When *amp_level* is set to `O0`, no cells and operators are converted.

When *amp_level* is set to `O1`, cells and operators in whitelist will be converted to lower precision operations. For details on whitelist, refer to *mindspore.amp.get_white_list()*.

When *amp_level* is set to `O2`, cells in blacklist will maintain full precision, and cells outside the list will be converted to low precision. For details on blacklist, refer to *mindspore.amp.get_black_list()*.

When *amp_level* is set to `O3`, all cells will be converted to low precision.

When *amp_level* is set to `auto`, operators in *auto_whitelist* will be converted to lower precision operations, operators in *auto_blacklist* will be converted to full precision operations, operators in *promote_list* will be converted to the higher accuracy float type of the operator inputs, and operators not listed will run in the type defined by their inputs.

Operators in *auto_whitelist* are:

`Conv2D`, `Conv3D`, `Conv2DTranspose`, `Conv3DTranspose`, `Convolution`, `MatMul`, `MatMulExt`, `BatchMatMul`, `BatchMatMulExt`, `PReLU`, `Einsum`, `Dense`, `Addmm`

Operators in *auto_blacklist* are:

`Pow`, `ACos`, `Asin`, `Cosh`, `Erfinv`, `Exp`, `Expm1`, `Log`, `Log1p`, `Reciprocal`, `Rsqrt`, `Sinh`, `Tan`, `Softplus`, `SoftplusExt`, `LayerNorm`, `LayerNormExt`, `BatchNorm`, `BatchNormExt`, `GroupNorm`, `KLDivLoss`, `SmoothL1Loss`, `MultilabelMarginLoss`, `SoftMarginLoss`, `TripletMarginLoss`, `MultiMarginLoss`, `BCEWithLogitsLoss`, `Pdist`, `Cdist`, `Renorm`, `ReduceProd`, `Softmax`, `LogSoftmax`, `CumProd`, `CumSum`, `CumsumExt`, `ProdExt`, `SumExt`, `Norm`, `MSELossExt`

Operators in *promote_list* are:

`Addcdiv`, `Addcmul`, `Cross`, `_PyboostCrossPrim`, `Dot`, `GridSampler2D`, `GridSampler3D`, `BiasAdd`, `AddN`, `Concat`

For details on automatic mixed precision, refer to Automatic Mix Precision .

Note:

- Repeatedly calling mixed-precision interfaces, such as *custom_mixed_precision* and *auto_mixed_precision*, can result in a larger network hierarchy and slower performance.
- If interfaces like *Model* and *build_train_network* is used to train the network which is converted by mixed-precision interfaces such as *custom_mixed_precision* and *auto_mixed_precision*, *amp_level* need to be configured to `O0` to avoid the duplicated accuracy conversion.
- When *amp_level* is set to `auto`, the output of the network may be lower precision. In this case, you may need to manually convert the type to avoid type inconsistency errors of the loss function.
- When *amp_level* is set to `auto`, and cells in the network are configured with *to_float*, the accuracy specified by *to_float* takes effect first.

Warning: `auto` level of *amp_level* is an experimental API that is subject to change or deletion.

Parameters

- **network** (*Union[Cell, function]*) –Definition of the network. Function type is supported only when *amp_level* is set to `auto`.
- **amp_level** (*str*) –Supports ["O0", "O1", "O2", "O3", "auto"]. Default: `"O0"`.
  - "O0": Do not change.
  - "O1": Convert cells and operators in whitelist to lower precision operations, and keep full precision operations for the rest.
  - "O2": Keep full precision operations for cells and operators in blacklist, and convert the rest to lower precision operations.
  - "O3": Cast network to lower precision.
  - "auto": Operators in *auto_whitelist* will be converted to lower precision operations, operators in *auto_blacklist* will be converted to full precision, operators in *promote_list* will be converted to the higher accuracy float type of the operator inputs, and operators not listed will run in the type defined by their inputs.
- **dtype** (*Type*) –The type used in lower precision calculations, can be `mstype.float16` or `mstype.bfloat16`, default: `mstype.float16`.

Raises

- **TypeError** –If *network* is not a Cell or a function.
- **ValueError** –If *dtype* is not one of `mstype.float16`, `mstype.bfloat16`.
- **ValueError** –If *amp_level* is not within the supported range.

Examples

```
>>> from mindspore import amp
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> network = LeNet5()
>>> amp_level = "O1"
>>> net = amp.auto_mixed_precision(network, amp_level)
```

6.2.2 mindspore.amp.build_train_network

`mindspore.amp.`**build_train_network**(*network*, *optimizer*, *loss_fn=None*, *level='O0'*, *boost_level='O0'*, ***kwargs*)

Build the mixed precision training cell automatically.

Note:

- After using *custom_mixed_precision* or *auto_mixed_precision* for precision conversion, it is not supported to perform the precision conversion again. If *build_train_network* is used to train a converted network, *level* need to be configured to `O0` to avoid the duplicated accuracy conversion.

Parameters

- **network** (Cell) −Definition of the network.
- **optimizer** (*mindspore.nn.Optimizer*) −Define the optimizer to update the Parameter.
- **loss_fn** (*Union[None, Cell]*) −Define the loss function. If None, the *network* should have the loss inside. Default: None .
- **level** (*str*) −Supports ['O0', 'O1', 'O2', 'O3', 'auto']. Default: 'O0' .

  For details on amp level, refer to *mindspore.amp.auto_mixed_precision()*.

  Property of *keep_batchnorm_fp32*, *cast_model_type* and *loss_scale_manager* determined by *level* setting may be overwritten by settings in *kwargs*.
- **boost_level** (*str*) −Option for argument *level* in *mindspore.boost* , level for boost mode training. Supports ['O0', 'O1', 'O2']. Default: 'O0' .
  - 'O0': Do not change.
  - 'O1': Enable the boost mode, the performance is improved by about 20%, and the accuracy is the same as the original accuracy.
  - 'O2': Enable the boost mode, the performance is improved by about 30%, and the accuracy is reduced by less than 3%.

  If 'O1' or 'O2' mode is set, the boost related library will take effect automatically.
- **cast_model_type** (*mindspore.dtype*) −Supports *mstype.float16* or *mstype.float32* . If set, the network will be casted to *cast_model_type* (*mstype.float16* or *mstype.float32*), but not to be casted to the type determined by *level* setting.
- **keep_batchnorm_fp32** (*bool*) −Keep Batchnorm run in *float32* when the network is set to cast to *float16* . If set, the *level* setting will take no effect on this property.
- **loss_scale_manager** (*Union[None, LossScaleManager]*) −If not None, must be subclass of *mindspore.amp.LossScaleManager* for scaling the loss. If set, the *level* setting will take no effect on this property.

Raises

ValueError −If device is CPU, property *loss_scale_manager* is not *None* or *mindspore.amp.FixedLossScaleManager* (with property *drop_overflow_update=False*).

Examples

```
>>> from mindspore import amp, nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> network = LeNet5()
>>> net_loss = nn.SoftmaxCrossEntropyWithLogits(reduction="mean")
>>> net_opt = nn.Momentum(network.trainable_params(), learning_rate=0.01, momentum=0.9)
>>> amp_level="O3"
>>> net = amp.build_train_network(network, net_opt, net_loss, amp_level)
```

6.2.3 mindspore.amp.custom_mixed_precision

mindspore.amp.**custom_mixed_precision**(*network*, *, *white_list=None*, *black_list=None*, *dtype=mstype.float16*)

When the *white_list* is provided, primitives and cells in *white_list* will perform the precision conversion. When the *black_list* is provided, cells that are not in *black_list* will perform the pereision conversion. Only one of *white_list* and *black_list* should be provided.

Note:

- Repeatedly calling mixed-precision interfaces, such as *custom_mixed_precision* and *auto_mixed_precision*, can result in a larger network hierarchy and slower performance.
- If interfaces like *Model* and *build_train_network* is used to train the network which is converted by mixed-precision interfaces such as *custom_mixed_precision* and *auto_mixed_precision*, *amp_level* or *level* need to be configured to `O0` to avoid the duplicated accuracy conversion.
- Primitives for blacklist is not support yet.

Parameters

network (`Cell`) – Definition of the network.

Keyword Arguments

- **white_list** (*list[Primitive, Cell], optional*) – White list of custom mixed precision. Defaults: `None` , means white list is not used.
- **black_list** (*list[Cell], optional*) – Black list of custom mixed precision. Defaults: `None` , means black list is not used.
- **dtype** (*Type*) – The type used in lower precision calculations, can be `mstype.float16` or `mstype.bfloat16` , default: `mstype.float16` .

Returns

network (Cell), A network supporting mixed precision.

Raises

- **TypeError** – The network type is not Cell.
- **ValueError** – Neither *white_list* nor *black_list* is provided.
- **ValueError** – If *dtype* is not one of `mstype.float16` , `mstype.bfloat16` .
- **ValueError** – Both *white_list* and *black_list* are provided.

Examples

```
>>> from mindspore import amp, nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> custom_white_list = amp.get_white_list()
>>> custom_white_list.append(nn.Flatten)
>>> net = amp.custom_mixed_precision(net, white_list=custom_white_list)
```

6.2.4 mindspore.amp.get_black_list

mindspore.amp.**get_black_list**()

Provide a copy of internal black list used by auto mixed precision with *amp_level* set to `O2`.

The current built-in blacklist contents are:

[*mindspore.nn.BatchNorm1d*, *mindspore.nn.BatchNorm2d*, *mindspore.nn.BatchNorm3d*, *mindspore.nn.LayerNorm*]

Returns

list, A copy of internal black list.

Examples

```
>>> from mindspore import amp
>>> black_list = amp.get_black_list()
>>> print(black_list)
[<class 'mindspore.nn.layer.normalization.BatchNorm1d'>, <class 'mindspore.nn.layer.
↪normalization.BatchNorm2d'>,
 <class 'mindspore.nn.layer.normalization.BatchNorm3d'>, <class 'mindspore.nn.layer.
↪normalization.LayerNorm'>]
```

6.2.5 mindspore.amp.get_white_list

mindspore.amp.**get_white_list**()

Provide a copy of internal white list used by auto mixed precision with *amp_level* set to `O1`.

The current built-in whitelist contents are:

[*mindspore.nn.Conv1d*, *mindspore.nn.Conv2d*, *mindspore.nn.Conv3d*, *mindspore.nn.Conv1dTranspose*, *mindspore.nn.Conv2dTranspose*, *mindspore.nn.Conv3dTranspose*, *mindspore.nn.Dense*, *mindspore.nn.LSTMCell*, *mindspore.nn.RNNCell*, *mindspore.nn.GRUCell*, *mindspore.ops.Conv2D*, *mindspore.ops.Conv3D*, *mindspore.ops.Conv2DTranspose*, *mindspore.ops.Conv3DTranspose*, *mindspore.ops.MatMul*, *mindspore.ops.BatchMatMul*, *mindspore.ops.PReLU*, *mindspore.ops.ReLU*, *mindspore.ops.Ger*]

Returns

list, A copy of internal white list.

Examples

```
>>> from mindspore import amp
>>> white_list = amp.get_white_list()
>>> print(white_list)
[<class 'mindspore.nn.layer.conv.Conv1d'>, <class 'mindspore.nn.layer.conv.Conv2d'>,
 <class 'mindspore.nn.layer.conv.Conv3d'>, <class 'mindspore.nn.layer.conv.Conv1dTranspose'>,
 <class 'mindspore.nn.layer.conv.Conv2dTranspose'>, <class 'mindspore.nn.layer.conv.
↪Conv3dTranspose'>,
 <class 'mindspore.nn.layer.basic.Dense'>, <class 'mindspore.nn.layer.rnn_cells.LSTMCell'>,
 <class 'mindspore.nn.layer.rnn_cells.RNNCell'>, <class 'mindspore.nn.layer.rnn_cells.GRUCell
↪'>,
 <class 'mindspore.ops.operations.nn_ops.Conv2D'>, <class 'mindspore.ops.operations.nn_ops.
↪Conv3D'>,
```

(continues on next page)

(continued from previous page)

```
 <class 'mindspore.ops.operations.nn_ops.Conv2DTranspose'>,
 <class 'mindspore.ops.operations.nn_ops.Conv3DTranspose'>,
 <class 'mindspore.ops.operations.nn_ops.Conv2DBackpropInput'>,
 <class 'mindspore.ops.operations.math_ops.MatMul'>, <class 'mindspore.ops.operations.math_
↪ops.BatchMatMul'>,
 <class 'mindspore.ops.operations.nn_ops.PReLU'>, <class 'mindspore.ops.operations.nn_ops.
↪ReLU'>,
 <class 'mindspore.ops.operations.math_ops.Ger'>]
```

6.3 Overflow Detection

mindspore.amp.all_finite	Returns a scalar Tensor indicating whether the inputs are finite.

6.3.1 mindspore.amp.all_finite

mindspore.amp.**all_finite**(*inputs*)

Returns a scalar Tensor indicating whether the inputs are finite.

> **Warning:** This is an experimental API that is subject to change or deletion.
>
> The interface must be used in whole network training scenario to detect whether grads are finite, and the results may be different on different device targets.

Parameters

inputs (*Union(tuple(Tensor), list(Tensor))*) – a iterable Tensor.

Returns

Tensor, a scalar Tensor and the dtype is bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import amp, Tensor
>>> import numpy as np
>>> x = (Tensor(np.array([np.log(-1), 1, np.log(0)])), Tensor(np.array([1.0])))
>>> output = amp.all_finite(x)
```

Tutorial Examples:

- Automatic Mix Precision - Loss Scaling

CHAPTER

SEVEN

MINDSPORE.TRAIN

7.1 Model

mindspore.train.Model	High-Level API for training or inference.

7.1.1 mindspore.train.Model

class mindspore.train.**Model**(*network*, *loss_fn=None*, *optimizer=None*, *metrics=None*, *eval_network=None*, *eval_indexes=None*, *amp_level='O0'*, *boost_level='O0'*, ***kwargs*)

High-Level API for training or inference.

Model groups layers into an object with training and inference features based on the arguments.

Note:

- If use mixed precision functions, need to set parameter *optimizer* at the same time, otherwise mixed precision functions do not take effect. When uses mixed precision functions, *global_step* in optimizer may be different from *cur_step_num* in Model.
- After using *custom_mixed_precision* or *auto_mixed_precision* for precision conversion, it is not supported to perform the precision conversion again. If *Model* is used to train a converted network, *amp_level* need to be configured to `O0` to avoid the duplicated accuracy conversion.

Parameters

- **network** (`Cell`) −A training or testing network.
- **loss_fn** (`Cell`) −Objective function. If *loss_fn* is None, the *network* should contain the calculation of loss. Default: `None` .
- **optimizer** (`Cell`) −Optimizer for updating the weights. If *optimizer* is None, the *network* needs to do backpropagation and update weights. Default: `None` .
- **metrics** (*`Union[dict, set]`*) −A Dictionary or a set of metrics for model evaluation. eg: {'accuracy', 'recall'}. Default: `None` .
- **eval_network** (`Cell`) −Network for evaluation. If not defined, *network* and *loss_fn* would be wrapped as *eval_network* . Default: `None` .
- **eval_indexes** (*`list`*) −It is used when eval_network is defined. If *eval_indexes* is None by default, all outputs of the *eval_network* would be passed to metrics. If *eval_indexes* is set, it must contain three elements: the positions of loss value, predicted value and label in outputs of the *eval_network*. In this case, the loss value will be passed

to the *Loss* metric, the predicted value and label will be passed to other metrics. `mindspore.train.Metric.set_indexes()` is recommended instead of *eval_indexes*. Default: `None` .

- **amp_level** (*str*) –Option for argument *level* in `mindspore.amp.build_train_network()`, level for mixed precision training. Supports ["O0", "O1", "O2", "O3", "auto"]. Default: `"O0"` .

  For details on *amp_level* , refer to `mindspore.amp.auto_mixed_precision()`.

  The BatchNorm strategy can be changed by *keep_batchnorm_fp32* settings in *kwargs*. *keep_batchnorm_fp32* must be a bool. The loss scale strategy can be changed by *loss_scale_manager* setting in *kwargs*. *loss_scale_manager* should be a subclass of `mindspore.amp.LossScaleManager`.

- **boost_level** (*str*) –Option for argument *level* in *mindspore.boost*, level for boost mode training. Supports ["O0", "O1", "O2"]. Default: `"O0"` .

  – "O0": Do not change.

  – "O1": Enable the boost mode, the performance is improved by about 20%, and the accuracy is the same as the original accuracy.

  – "O2": Enable the boost mode, the performance is improved by about 30%, and the accuracy is reduced by less than 3%.

  If you want to config boost mode by yourself, you can set boost_config_dict as *boost.py*. In order for this function to work, you need to set the parameter *optimizer*, along with at least one of the parameter *eval_network* or performance *metrics*.

  Notice: The current optimization enabled by default only applies to some networks, and not all networks can obtain the same benefits. It is recommended to enable this function on the Graph mode + Ascend platform, and for better acceleration, refer to `mindspore.boost.AutoBoost` to configure boost_config_dict.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics=None)
>>> model.train_network
>>> model.predict_network
>>> model.eval_network
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> model.train(2, dataset)
```

build(*train_dataset=None*, *valid_dataset=None*, *sink_size=- 1*, *epoch=1*, *sink_mode=True*)

Build computational graphs and data graphs with the sink mode.

> **Warning:** This is an experimental API that is subject to change or deletion.

Note: The interface builds the computational graphs, when the interface is executed first, 'Model.train' only performs the

graphs execution. Pre-build process only supports *GRAPH_MODE* and *Ascend* target currently. It only supports dataset sink mode.

Parameters

- **train_dataset** (*Dataset*) – A training dataset iterator. If *train_dataset* is defined, training graphs will be built. Default: `None` .
- **valid_dataset** (*Dataset*) – An evaluating dataset iterator. If *valid_dataset* is defined, evaluation graphs will be built, and *metrics* in *Model* can not be None. Default: `None` .
- **sink_size** (*int*) – Control the number of steps for each sinking. Default: `-1` .
- **epoch** (*int*) – Control the training epochs. Default: `1` .
- **sink_mode** (*bool*) – Determines whether to pass the data through dataset channel. Default: `True` .

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model
>>> from mindspore.amp import FixedLossScaleManager
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> loss_scale_manager = FixedLossScaleManager()
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics=None,
...                   loss_scale_manager=loss_scale_manager)
>>> model.build(dataset, epoch=2)
>>> model.train(2, dataset)
```

eval(*valid_dataset*, *callbacks=None*, *dataset_sink_mode=False*)

Evaluation API.

Configure to pynative mode or CPU, the evaluating process will be performed with dataset non-sink mode.

Note: If dataset_sink_mode is True, data will be sent to device. At this point, the dataset will be bound to this model, so the dataset cannot be used by other models. If the device is Ascend, features of data will be transferred one by one. The limitation of data transmission per time is 256M.

The interface builds the computational graphs and then executes the computational graphs. However, when the *Model.build* is executed first, it only performs the graphs execution.

Parameters

- **valid_dataset** (*Dataset*) – Dataset to evaluate the model.
- **callbacks** (*Optional[list(Callback), Callback]*) – List of callback objects or callback object, which should be executed while evaluation. Default: `None` .

- **dataset_sink_mode** (*bool*) –Determines whether to pass the data through dataset channel. Default: `False` .

Returns

Dict, the key is the metric name defined by users and the value is the metrics value for the model in the test mode.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> model = Model(net, loss_fn=loss, optimizer=None, metrics={'acc'})
>>> acc = model.eval(dataset, dataset_sink_mode=False)
```

property eval_network

Get the model's eval network.

Returns

Object, the instance of evaluate network.

fit(*epoch*, *train_dataset*, *valid_dataset=None*, *valid_frequency=1*, *callbacks=None*, *dataset_sink_mode=False*, *valid_dataset_sink_mode=False*, *sink_size=- 1*, *initial_epoch=0*)

Fit API.

Evaluation process will be performed during training process if *valid_dataset* is provided.

More details please refer to *mindspore.train.Model.train()* and *mindspore.train.Model.eval()*.

Parameters

- **epoch** (*int*) –Total training epochs. Generally, train network will be trained on complete dataset per epoch. If *dataset_sink_mode* is set to True and *sink_size* is greater than 0, each epoch will train *sink_size* steps instead of total steps of dataset. If *epoch* used with *initial_epoch*, it is to be understood as "final epoch".
- **train_dataset** (*Dataset*) –A training dataset iterator. If *loss_fn* is defined, the data and label will be passed to the *network* and the *loss_fn* respectively, so a tuple (data, label) should be returned from dataset. If there is multiple data or labels, set *loss_fn* to None and implement calculation of loss in *network*, then a tuple (data1, data2, data3, ⋯) with all data returned from dataset will be passed to the *network*.
- **valid_dataset** (*Dataset*) –Dataset to evaluate the model. If *valid_dataset* is provided, evaluation process will be performed on the end of training process. Default: `None` .
- **valid_frequency** (*int, list*) –Only relevant if *valid_dataset* is provided. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. *valid_frequency=2* runs validation every 2 epochs. If a list, specifies the epochs on which to run validation, e.g. *valid_frequency=[1, 5]* runs validation at the end of the 1st, 5th epochs. Default: `1` .
- **callbacks** (*Optional[list[Callback], Callback]*) –List of callback objects or callback object, which should be executed while training. Default: `None` .
- **dataset_sink_mode** (*bool*) –Determines whether to pass the train data through dataset channel. Configure pynative mode or CPU, the training process will be performed with dataset not sink. Default: `False` .

- **valid_dataset_sink_mode** (*bool*) –Determines whether to pass the validation data through dataset channel. Default: `False` .
- **sink_size** (*int*) –Control the number of steps for each sinking. *sink_size* is invalid if *dataset_sink_mode* is False. If sink_size = -1, sink the complete dataset for each epoch. If sink_size > 0, sink sink_size data for each epoch. Default: `-1` .
- **initial_epoch** (*int*) –Epoch at which to start train, it useful for resuming a previous training run. Default: `0` .

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> train_dataset = create_dataset("train")
>>> valid_dataset = create_dataset("test")
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics={"accuracy"})
>>> model.fit(2, train_dataset, valid_dataset)
```

infer_predict_layout(**predict_data*, *skip_backend_compile=False*)

Generate parameter layout for the predict network when using *AutoParallel(cell)* to enable parallel mode.

Data could be a single tensor or multiple tensors.

Note: Batch data should be put together in one tensor.

Parameters

- **predict_data** (*Union[Tensor, list[Tensor], tuple[Tensor]], optional*) –The predict data, can be a single tensor, a list of tensor, or a tuple of tensor.
- **skip_backend_compile** (*bool*) –Only run the frontend compile process, skip the compile process on the device side. Set this flag to True may lead to recompiling process can not hit cache.

Returns

Dict, Parameter layout dictionary used for load distributed checkpoint. Using as one of input parameters of load_distributed_checkpoint, always.

Raises

RuntimeError –If not in GRAPH_MODE.

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore.train import Model
>>> from mindspore.ops import operations as P
>>> from mindspore import context
>>> from mindspore.communication import init
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>>
>>> class Net(nn.Cell):
>>>     def __init__(self):
>>>         super(Net, self).__init__()
>>>         self.fc1 = nn.Dense(128, 768, activation='relu')
>>>         self.fc2 = nn.Dense(128, 768, activation='relu')
>>>         self.fc3 = nn.Dense(128, 768, activation='relu')
>>>         self.fc4 = nn.Dense(768, 768, activation='relu')
>>>         self.relu4 = nn.ReLU()
>>>         self.relu5 = nn.ReLU()
>>>         self.transpose = P.Transpose()
>>>         self.matmul1 = P.MatMul()
>>>         self.matmul2 = P.MatMul()
>>>
>>>     def construct(self, x):
>>>         q = self.fc1(x)
>>>         k = self.fc2(x)
>>>         v = self.fc3(x)
>>>         k = self.transpose(k, (1, 0))
>>>         c = self.relu4(self.matmul1(q, k))
>>>         s = self.relu5(self.matmul2(c, v))
>>>         s = self.fc4(s)
>>>         return s
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> inputs = Tensor(np.ones([32, 128]).astype(np.float32))
>>> net = Net()
>>> parallel_net = AutoParallel(net, parallel_mode='semi_auto')
>>> model = Model(parallel_net)
>>> predict_map = model.infer_predict_layout(inputs)
```

infer_train_layout(*train_dataset*, *dataset_sink_mode=True*, *sink_size=-1*)

Generate parameter layout for the train network when using *AutoParallel(cell)* to enable parallel mode.

Only dataset sink mode is supported for now.

> **Warning:** This is an experimental API that is subject to change or deletion.

Note: This is a pre-compile function. The arguments should be the same as model.train() function.

Parameters

- **train_dataset** (*Dataset*) –A training dataset iterator. If there is no loss_fn, a tuple with multiple data

(data1, data2, data3, ···) should be returned and passed to the network. Otherwise, a tuple (data, label) should be returned. The data and label would be passed to the network and loss function respectively.

- **dataset_sink_mode** (*bool*) –Determines whether to pass the data through dataset channel. Configure pynative mode or CPU, the training process will be performed with dataset not sink. Default: `True` .
- **sink_size** (*int*) –Control the number of steps for each sinking. If dataset_sink_mode is False, set sink_size as invalid. If sink_size = -1, sink the complete dataset for each epoch. If sink_size > 0, sink sink_size data for each epoch. Default: `-1` .

Returns

Dict, Parameter layout dictionary used for load distributed checkpoint

Examples

```
>>> # This example should be run with multiple devices. Refer to the tutorial >␣
↪Distributed Training on
>>> # mindspore.cn.
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn
>>> from mindspore.train import Model
>>> from mindspore.communication import init
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> parallel_net = AutoParallel(net)
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> loss_scale_manager = ms.FixedLossScaleManager()
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(parallel_net, loss_fn=loss, optimizer=optim, metrics=None,
...                   loss_scale_manager=loss_scale_manager)
>>> layout_dict = model.infer_train_layout(dataset)
```

predict(**predict_data*, *backend=None*, *config=None*)

Generate output predictions for the input samples.

Parameters

- **predict_data** (*Union[Tensor, list[Tensor], tuple[Tensor]], optional*) –The predict data, can be a single tensor, a list of tensor, or a tuple of tensor.
- **backend** (*str*) –Select predict backend, this parameter is an experimental feature and is mainly used for MindSpore Lite cloud-side inference. Default: `None` .
- **config** (*dict, optional*) –The config includes two parts: config_path (configPath, str) and config_item (str, dict). When the config_item is set, its priority is higher than the config_path. Set the ranking table file for inference. The content of the configuration file is as follows:

config_path defines the path of the configuration file, which is used to pass user-defined options during model building. In the following scenarios, users may need to set parameters. For example: "/home/user/config.ini". Default value: `""` , here is the content of the config.ini file:

```
[ascend_context]
rank_table_file = [path_a](storage initial path of the rank table file)
[execution_plan]
[op_name1] = data_type:float16 (operator named op_name1 is set to data type␣
↪float16)
[op_name2] = data_type:float32 (operator named op_name2 is set to data type␣
↪float32)
```

When only the config_path is configured, it is done as follows:

```
config = {"configPath" : "/home/user/config.ini"}
```

When only the config_dict is configured, it is done as follows:

```
config = {"ascend_context" : {"rank_table_file" : "path_b"},
          "execution_plan" : {"op_name1" : "data_type:float16", "op_name2" :
↪"data_type:float32"}}
```

When both the *config_path* and the *config_dict* are configured, it is done as follows:

```
config = {"configPath" : "/home/user/config.ini",
          "ascend_context" : {"rank_table_file" : "path_b"},
          "execution_plan" : {"op_name3" : "data_type:float16", "op_name4" :
↪"data_type:float32"}}
```

Note that both the "configPath" is configured in the config_dict and the config_item, in this case, the path_b in the config_dict takes precedence.

Returns

Tensor, array(s) of predictions.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import Model
>>>
>>> input_data = Tensor(np.random.randint(0, 255, [1, 1, 32, 32]), mindspore.float32)
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> model = Model(LeNet5())
>>> result = model.predict(input_data)
```

property predict_network

Get the model's predict network.

Returns

Object, the instance of predict network.

train(*epoch*, *train_dataset*, *callbacks=None*, *dataset_sink_mode=False*, *sink_size=- 1*, *initial_epoch=0*)

Training API.

When setting pynative mode or CPU, the training process will be performed with dataset not sink.

Note: If dataset_sink_mode is True, data will be sent to device. If the device is Ascend, features of data will be transferred one by one. The limitation of data transmission per time is 256M.

When dataset_sink_mode is True, the *step_end* method of the instance of Callback will be called at the end of step in PyNative mode, or will be called at the end of epoch in Graph mode.

If dataset_sink_mode is True, dataset will be bound to this model and cannot be used by other models.

If sink_size > 0, each epoch of the dataset can be traversed unlimited times until you get sink_size elements of the dataset. The next epoch continues to traverse from the end position of the previous traversal.

The interface builds the computational graphs and then executes the computational graphs. However, when the *Model.build* is executed first, it only performs the graphs execution.

Parameters

- **epoch** (*int*) –Total training epochs. Generally, train network will be trained on complete dataset per epoch. If *dataset_sink_mode* is set to True and *sink_size* is greater than 0, each epoch will train *sink_size* steps instead of total steps of dataset. If *epoch* used with *initial_epoch*, it is to be understood as "final epoch".
- **train_dataset** (*Dataset*) –A training dataset iterator. If *loss_fn* is defined, the data and label will be passed to the *network* and the *loss_fn* respectively, so a tuple (data, label) should be returned from dataset. If there is multiple data or labels, set *loss_fn* to None and implement calculation of loss in *network*, then a tuple (data1, data2, data3, ⋯) with all data returned from dataset will be passed to the *network*.
- **callbacks** (*Optional[list[Callback], Callback]*) –List of callback objects or callback object, which should be executed while training. Default: `None`.
- **dataset_sink_mode** (*bool*) –Determines whether to pass the data through dataset channel. Configure pynative mode or CPU, the training process will be performed with dataset not sink. Default: `False`.
- **sink_size** (*int*) –Control the number of steps for each sinking. *sink_size* is invalid if *dataset_sink_mode* is False. If sink_size = -1, sink the complete dataset for each epoch. If sink_size > 0, sink sink_size data for each epoch. Default: -1.
- **initial_epoch** (*int*) –Epoch at which to start train, it used for resuming a previous training run. Default: 0.

Examples

```
>>> import mindspore as ms
>>> from mindspore import nn
>>> from mindspore.train import Model
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> loss_scale_manager = ms.FixedLossScaleManager(1024., False)
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics=None,
...                   loss_scale_manager=loss_scale_manager)
>>> model.train(2, dataset)
```

property train_network
Get the model's train network.

Returns

Object, the instance of train network.

7.2 Callback

mindspore.train.BackupAndRestore	Callback to back up and restore the parameters during training.
mindspore.train.Callback	Abstract base class used to build a Callback class.
mindspore.train.CheckpointConfig	The configuration of model checkpoint.
mindspore.train.EarlyStopping	Stop training when a monitored metric has stopped improving.
mindspore.train.FlopsUtilizationCollector	The FlopsUtilizationCollector interface counts the model utilization information MFU and the hardware utilization information HFU.
mindspore.train.History	Records the network outputs and metrics information into a *History* object.
mindspore.train.LambdaCallback	Callback for creating simple, custom callbacks.
mindspore.train.LearningRateScheduler	Change the learning_rate during training.
mindspore.train.LossMonitor	Monitor the loss in train or monitor the loss and eval metrics in fit.
mindspore.train.TrainFaultTolerance	This callback is used to enable the TFT feature MindIO TFT and will execute TFT operations during training process, such as TFT init, report and exception handle.
mindspore.train.ModelCheckpoint	The checkpoint callback class.
mindspore.train.OnRequestExit	Respond to the user's closing request, exit the training or eval process, and save the checkpoint and mindir.
mindspore.train.ReduceLROnPlateau	Reduce learning rate when the monitor has stopped improving.
mindspore.train.RunContext	Hold and manage information about the model.
mindspore.train.TimeMonitor	Monitor the time in train or eval process.

7.2.1 mindspore.train.BackupAndRestore

class mindspore.train.**BackupAndRestore**(*backup_dir*, *save_freq='epoch'*, *delete_checkpoint=True*)
Callback to back up and restore the parameters during training.

Note: This function can only use in training.

Parameters

- **backup_dir** (*str*) – Path to store and load the checkpoint file.
- **save_freq** (*Union["epoch", int]*) – When set to `"epoch"` the callback saves the checkpoint at the end of each epoch. When set to an integer, the callback saves the checkpoint every *save_freq* epoch. Default: `"epoch"` .
- **delete_checkpoint** (*bool*) – If *delete_checkpoint=True*, the checkpoint will be deleted after training is finished. Default: `True` .

Raises

- **ValueError** –If backup_dir is not str.
- **ValueError** –If save_freq is not "epoch" or int.
- **ValueError** –If delete_checkpoint is not bool.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, BackupAndRestore, RunContext
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> backup_ckpt = BackupAndRestore("backup")
>>> model.train(10, dataset, callbacks=backup_ckpt)
```

on_train_begin(*run_context*)

Load the backup checkpoint file at the beginning of epoch.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

on_train_end(*run_context*)

Deleted checkpoint file at the end of train.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

on_train_epoch_end(*run_context*)

Backup checkpoint file at the end of train epoch.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

7.2.2 mindspore.train.Callback

class mindspore.train.**Callback**

Abstract base class used to build a Callback class. Callbacks are context managers which will be entered and exited when passing into the Model. You can use this mechanism to do some custom operations.

Each method of Callback class corresponds to a stage in training or eval process, and those methods have the same input *run_context*, which hold context information of the model in training or eval process. When defining a Callback subclass or creating a custom Callback, note that you should override methods with names prefixed with "on_train" or "on_eval", otherwise ValueError will be raised if the custimized Callbacks used in *model.fit*.

When creating a custom Callback, model context information can be obtained in Callback methods by calling *RunContext.original_args()*, which is a dictionary varivable recording current attributes. Users can add custimized attributes to the information. Training process can also be stopped by calling *request_stop* method.

Examples

```
>>> import numpy as np
>>> from mindspore import nn
>>> from mindspore import dataset as ds
>>> from mindspore.train import Model, Callback
>>> class Print_info(Callback):
...     def step_end(self, run_context):
...         cb_params = run_context.original_args()
...         print("step_num: ", cb_params.cur_step_num)
>>>
>>> print_cb = Print_info()
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> net = nn.Dense(10, 5)
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> model.train(1, dataset, callbacks=print_cb)
step_num: 1
step_num: 2
```

begin(*run_context*)

Called once before the network executing. A backwards compatibility alias for *on_train_begin* and *on_eval_begin*.

Note: *begin* is deprecated and will be deleted in a future version, please use *on_train_begin* and *on_eval_begin* instead.

Parameters

run_context (RunContext) – Include some information of the model.

end(*run_context*)

Called once after network training. A backwards compatibility alias for *on_train_end* and *on_eval_end*.

Note: *end* is deprecated and will be deleted in a future version, please use *on_train_end* and *on_eval_end* instead.

Parameters

run_context (RunContext) – Include some information of the model.

epoch_begin(*run_context*)

Called before each epoch beginning. A backwards compatibility alias for *on_train_epoch_begin* and *on_eval_epoch_begin*.

Note: *epoch_begin* is deprecated and will be deleted in a future version, please use *on_train_epoch_begin* and *on_eval_epoch_begin* instead.

Parameters

run_context (RunContext) –Include some information of the model.

epoch_end(*run_context*)
Called after each epoch finished. A backwards compatibility alias for *on_train_epoch_end* and *on_eval_epoch_end*.

Note: *epoch_end* is deprecated and will be deleted in a future version, please use *on_train_epoch_end* and *on_eval_epoch_end* instead.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_begin(*run_context*)
Called before eval begin.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_end(*run_context*)
Called after eval end.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_epoch_begin(*run_context*)
Called before eval epoch begin.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_epoch_end(*run_context*)
Called after eval epoch end.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_step_begin(*run_context*)
Called before each eval step begin.

Parameters

run_context (RunContext) –Include some information of the model.

on_eval_step_end(*run_context*)
Called after each eval step end.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_begin(*run_context*)
Called once before the network training.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_end(*run_context*)
Called after training end.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_epoch_begin(*run_context*)
Called before each training epoch begin.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_epoch_end(*run_context*)
Called after each training epoch end.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_step_begin(*run_context*)
Called before each training step begin.

Parameters

run_context (RunContext) –Include some information of the model.

on_train_step_end(*run_context*)
Called after each training step end.

Parameters

run_context (RunContext) –Include some information of the model.

step_begin(*run_context*)
Called before each step beginning. A backwards compatibility alias for *on_train_step_begin* and *on_eval_step_begin*.

Note: *step_begin* is deprecated and will be deleted in a future version, please use *on_train_step_begin* and *on_eval_step_begin* instead.

Parameters

run_context (RunContext) –Include some information of the model.

step_end(*run_context*)
Called after each step finished. A backwards compatibility alias for *on_train_step_end* and *on_eval_step_end*.

Note: *step_end* is deprecated and will be deleted in a future version, please use *on_train_step_end* and *on_eval_step_end* instead.

Parameters

run_context (RunContext) –Include some information of the model.

7.2.3 mindspore.train.CheckpointConfig

class mindspore.train.**CheckpointConfig**(*save_checkpoint_steps=1*, *save_checkpoint_seconds=0*, *keep_checkpoint_max=5*, *keep_checkpoint_per_n_minutes=0*, *integrated_save=True*, *async_save=False*, *saved_network=None*, *append_info=None*, *enc_key=None*, *enc_mode='AES-GCM'*, *exception_save=False*, *crc_check=False*, *remove_redundancy=False*, *format='ckpt'*, ***kwargs*)

The configuration of model checkpoint.

Note:

- During the training process, if dataset is transmitted through the data channel, it is suggested to set 'save_checkpoint_steps' to an integer multiple of loop_size. Otherwise, the time to save the checkpoint may be biased. It is recommended to set only one save strategy and one keep strategy at the same time. If both *save_checkpoint_steps* and *save_checkpoint_seconds* are set, *save_checkpoint_seconds* will be invalid. If both *keep_checkpoint_max* and *keep_checkpoint_per_n_minutes* are set, *keep_checkpoint_per_n_minutes* will be invalid.
- The *enc_mode* and *crc_check* parameters are mutually exclusive and cannot be configured simultaneously.

Parameters

- **save_checkpoint_steps** (*int*) −Steps to save checkpoint. Default: `1` .
- **save_checkpoint_seconds** (*int*) −Seconds to save checkpoint. Can't be used with save_checkpoint_steps at the same time. Default: `0` .
- **keep_checkpoint_max** (*int*) −Maximum number of checkpoint files can be saved. Default: `5` .
- **keep_checkpoint_per_n_minutes** (*int*) −Save the checkpoint file every *keep_checkpoint_per_n_minutes* minutes. Can't be used with keep_checkpoint_max at the same time. Default: `0` .
- **integrated_save** (*bool*) −Whether to merge and save the split Tensor in the automatic parallel scenario. Integrated save function is only supported in automatic parallel scene, not supported in manual parallel. Default: `True` .
- **async_save** (*Union[bool, str], optional*) −Whether to use asynchronous saving of the checkpoint file or safetensors file, if True, the asynchronous thread is used by default. If the type is string, the method of asynchronous saving, it can be "process" or "thread". Default: `False` .
- **saved_network** (`Cell`) −Network to be saved in checkpoint file. If the saved_network has no relation with the network in training, the initial value of saved_network will be saved. Default: `None` .
- **append_info** (*list*) −The information save to checkpoint file. Support "epoch_num", "step_num" and dict. The key of dict must be str, the value of dict must be one of int, float, bool, Parameter or Tensor. Default: `None` .
- **enc_key** (*Union[None, bytes]*) −Byte type key used for encryption. If the value is None, the encryption is not required. Default: `None` .
- **enc_mode** (*str*) −This parameter is valid only when enc_key is not set to None. Specifies the encryption mode, currently supports 'AES-GCM', 'AES-CBC' and 'SM4-CBC'. Default: `'AES-GCM'` .
- **exception_save** (*bool*) −Whether to save the current checkpoint when an exception occurs. Default: `False` .
- **crc_check** (*bool*) −Whether to perform crc32 calculation when saving checkpoint and save the calculation result to the end of ckpt. Default: `False` .
- **remove_redundancy** (*bool*) −Whether to enable saving the checkpoint with redundancy removal. Redundancy removal refers to eliminating redundant data in data parallelism mode. Default: `False` , means redundant-free saving is not enabled.

- **format** (*str*) –Format of the output file, can be "ckpt" or "safetensors". Default: "ckpt".
- **kwargs** (*dict*) –Configuration options dictionary.

Raises

ValueError –If input parameter is not the correct type.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, CheckpointConfig, ModelCheckpoint
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> config = CheckpointConfig(save_checkpoint_seconds=100, keep_checkpoint_per_n_minutes=5,␣
↪saved_network=net)
>>> config.save_checkpoint_steps
1
>>> config.save_checkpoint_seconds
>>> config.keep_checkpoint_max
5
>>> config.keep_checkpoint_per_n_minutes
>>> config.integrated_save
True
>>> config.async_save
False
>>> config.saved_network
>>> config.enc_key
>>> config.enc_mode
'AES-GCM'
>>> config.append_dict
>>> config.get_checkpoint_policy
>>> ckpoint_cb = ModelCheckpoint(prefix='LeNet5', directory='./checkpoint', config=config)
>>> model.train(10, dataset, callbacks=ckpoint_cb)
```

property append_dict

Get the value of information dict saved to checkpoint file.

Returns

dict, the information saved to checkpoint file.

property async_save

Get the value of whether or how asynchronous execution saves the checkpoint to a file.

Returns

(bool, str), whether or how asynchronous execution saves the checkpoint to a file.

property crc_check

Get the value of the whether to enable crc check.

Returns

bool, whether to enable crc check.

property enc_key
Get the value of byte type key used for encryption.

Returns

(None, bytes), byte type key used for encryption.

property enc_mode
Get the value of the encryption mode.

Returns

str, encryption mode.

get_checkpoint_policy()
Get the policy of checkpoint.

Returns

dict, the information of checkpoint policy.

property integrated_save
Get the value of whether to merge and save the split Tensor in the automatic parallel scenario.

Returns

bool, whether to merge and save the split Tensor in the automatic parallel scenario.

property keep_checkpoint_max
Get the value of maximum number of checkpoint files can be saved.

Returns

int, Maximum number of checkpoint files can be saved.

property keep_checkpoint_per_n_minutes
Get the value of save the checkpoint file every n minutes.

Returns

Int, save the checkpoint file every n minutes.

property map_param_inc
Get the value of whether to save map Parameter incrementally.

Returns

bool, whether to save map Parameter incrementally.

property save_checkpoint_seconds
Get the value of _save_checkpoint_seconds.

Returns

int, seconds to save the checkpoint file.

property save_checkpoint_steps
Get the value of steps to save checkpoint.

Returns

int, steps to save checkpoint.

property saved_network
Get the value of network to be saved in checkpoint file.

Returns

Cell, network to be saved in checkpoint file.

7.2.4 mindspore.train.EarlyStopping

class mindspore.train.**EarlyStopping**(*monitor='eval_loss'*, *min_delta=0*, *patience=0*, *verbose=False*, *mode='auto'*, *baseline=None*, *restore_best_weights=False*)

Stop training when a monitored metric has stopped improving.

Assuming *monitor* is "accuracy", with this, *mode* would be "max" since goal of trianing is to maximize the accuracy, the *model.fit()* training loop will check at end of epoch whether the accuracy is no longer increasing, considering the *min_delta* and *patience* if applicable. Once it's found no longer increasing, *run_context.request_stop()* will be called and the training terminates.

Parameters

- **monitor** (*str*) –quantity to be monitored. If evaluation is performed on the end of train epochs, the valid monitors can be "loss", "eval_loss" or metric names passed when instantiate the *Model*; otherwise the valid monitor is "loss". When monitor is "loss", if train network has multiple outputs, the first element will be returned as training loss. Default: `'eval_loss'` .
- **patience** (*int*) –*monitor* value is better than history best value over *min_delta* is seen as improvement, *patience* is number of epochs with no improvement that would be waited. When the waiting counter *self.wait* is larger than or equal to *patience*, the training process will be stopped. Default: `0` .
- **verbose** (*bool*) –If False: quiet, if True: print related information. Default: `False` .
- **mode** (*str*) –one of *{'auto', 'min', 'max'}*. In "min" mode, the learning rate will be reduced when the quantity monitored has stopped decreasing; in "max" mode it will be reduced when the quantity monitored has stopped increasing; in "auto" mode, the direction is automatically inferred from the name of the monitored quantity. Default: `'auto'` .
- **min_delta** (*float*) –threshold for measuring the new optimum, to only focus on significant changes. Default: `0` .
- **baseline** (*float*) –Baseline value for the monitor. When the monitor value shows improvement over the history best value and the baseline, the internal wait counter will be set to zero. Default: `None` .
- **restore_best_weights** (*bool*) –Whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used. Default: `False` .

Raises

- **ValueError** –*mode* not in 'auto', 'min' or 'max'.
- **ValueError** –The monitor value is not a scalar.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, EarlyStopping
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics={"acc"})
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
```

(continues on next page)

(continued from previous page)

```
>>> cb = EarlyStopping(monitor="acc", patience=3, verbose=True)
>>> model.fit(10, dataset, callbacks=cb)
```

on_train_begin(*run_context*)
Initialize variables at the begin of training.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_end(*run_context*)
If verbose is True, print the stopped epoch.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_epoch_end(*run_context*)
monitors the training process and if no improvement is seen for a 'patience' number of epochs, the training process will be stopped.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

7.2.5 mindspore.train.FlopsUtilizationCollector

class mindspore.train.**FlopsUtilizationCollector**(*data_size*, *computility=1*, *full_flops=True*, *enable_ma_collector=False*)

The FlopsUtilizationCollector interface counts the model utilization information MFU and the hardware utilization information HFU. Currently, the API counts only the forward and backward flops of MatMul, BatchMatMul, FlashAttentionScore, and Conv2D operators. Only used in graph mode with static shape.

Parameters

- **data_size** (*int*) –How many steps are the intervals between print information each time.
- **computility** (*int*) –The peak flops of each compute card. Default: `1` .
- **full_flops** (*bool*) –Whether to count the full model flops. If set full_flops to False, FlopsUtilizationCollector would count the shard model flops in each device. Default: `True` .
- **enable_ma_collector** (*bool*) –Whether to write flops into the log and provide them to tasks on the cloud for retrieval. Default: `False` .

Raises

- **TypeError** –If data_size is not positive int.
- **TypeError** –If full_flops is not bool.
- **TypeError** –If enable_ma_collector is not bool.
- **AssertionError** –If the training mode is not a static graph or not a static shape.

Examples

```
>>> import numpy as np
>>> import mindspore.dataset as ds
>>> from mindspore import nn
>>> from mindspore.train import Model, FlopsUtilizationCollector
>>> from mindspore import context
>>> context.set_context(mode=context.GRAPH_MODE)
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> net = nn.Dense(10, 5)
>>> crit = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> opt = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> flops_callback = FlopsUtilizationCollector(train_dataset.get_dataset_size(),␣
↪computility=10e6)
>>> model = Model(network=net, optimizer=opt, loss_fn=crit, metrics={"recall"})
>>> model.train(2, train_dataset, callbacks=[flops_callback])
Full model flops is 6400, Full hardware flops is 6400, Shard model flops is 6400, Shard␣
↪hardware flops is 6400
Train per step time: 135.572 ms, mfu:0.47% hfu:0.47%
Train per step time: 1.317 ms, mfu:48.59% hfu:48.59%
```

step_begin(*run_context*)
Record time at the beginning of step.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

step_end(*run_context*)
Print mfu and hfu time at the end of step.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

7.2.6 mindspore.train.History

class mindspore.train.**History**
Records the network outputs and metrics information into a *History* object.

- The network outputs information will be the loss value if not custimizing the train network or eval network;
- If the train network or eval network is custimized:
  - if the custimized network returns a *Tensor* or *numpy.ndarray*, the mean value of network output will be recorded.
  - if the custimized network returns a *tuple* or *list*, the first element of network outputs will be recorded.

Note: Normally used in *mindspore.train.Model.train()* or *mindspore.train.Model.fit()*.

Examples

```
>>> import numpy as np
>>> import mindspore.dataset as ds
>>> from mindspore import nn
>>> from mindspore.train import Model, History
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> net = nn.Dense(10, 5)
>>> crit = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> opt = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> history_cb = History()
>>> model = Model(network=net, optimizer=opt, loss_fn=crit, metrics={"recall"})
>>> model.train(2, train_dataset, callbacks=[history_cb])
>>> print(history_cb.epoch)
{'epoch': [1, 2]}
>>> print(history_cb.history)
{'net_output': [1.607877, 1.6033841]}
```

begin(*run_context*)

Initialize the *epoch* property at the begin of training.

Parameters

run_context (RunContext) – Context of the *mindspore.train.Model.{train | eval}*. For more details, please refer to *mindspore.train.RunContext*.

epoch_end(*run_context*)

Records the first element of network outputs and metrics information at the end of epoch.

Parameters

run_context (RunContext) – Context of the *mindspore.train.Model.{train | eval}*. For more details, please refer to *mindspore.train.RunContext*.

7.2.7 mindspore.train.LambdaCallback

class mindspore.train.**LambdaCallback**(*on_train_epoch_begin=None*, *on_train_epoch_end=None*, *on_train_step_begin=None*, *on_train_step_end=None*, *on_train_begin=None*, *on_train_end=None*, *on_eval_epoch_begin=None*, *on_eval_epoch_end=None*, *on_eval_step_begin=None*, *on_eval_step_end=None*, *on_eval_begin=None*, *on_eval_end=None*)

Callback for creating simple, custom callbacks.

This callback is constructed with anonymous functions that will be called at the appropriate time (during *mindspore.train.Model.{train | eval | fit}*). Note that each stage of callbacks expects one positional arguments: *run_context*.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **on_train_epoch_begin** (*Function*) – called at each train epoch begin. Default: None .
- **on_train_epoch_end** (*Function*) – called at each train epoch end. Default: None .
- **on_train_step_begin** (*Function*) – called at each train step begin. Default: None .

- **on_train_step_end** (*Function*) –called at each train step end. Default: None .
- **on_train_begin** (*Function*) –called at the beginning of model train. Default: None .
- **on_train_end** (*Function*) –called at the end of model train. Default: None .
- **on_eval_epoch_begin** (*Function*) –called at each eval epoch begin. Default: None .
- **on_eval_epoch_end** (*Function*) –called at each eval epoch end. Default: None .
- **on_eval_step_begin** (*Function*) –called at each eval step begin. Default: None .
- **on_eval_step_end** (*Function*) –called at each eval step end. Default: None .
- **on_eval_begin** (*Function*) –called at the beginning of model eval. Default: None .
- **on_eval_end** (*Function*) –called at the end of model eval. Default: None .

Examples

```
>>> import numpy as np
>>> import mindspore.dataset as ds
>>> from mindspore import nn
>>> from mindspore.train import Model, LambdaCallback
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> net = nn.Dense(10, 5)
>>> crit = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> opt = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> lambda_callback = LambdaCallback(on_train_epoch_end=
... lambda run_context: print("loss:", run_context.original_args().net_outputs))
>>> model = Model(network=net, optimizer=opt, loss_fn=crit, metrics={"recall"})
>>> model.train(2, train_dataset, callbacks=[lambda_callback])
loss: 1.6127687
loss: 1.6106578
```

7.2.8 mindspore.train.LearningRateScheduler

class mindspore.train.**LearningRateScheduler**(*learning_rate_function*)

Change the learning_rate during training.

Parameters

learning_rate_function (*Function*) –The function about how to change the learning rate during training.

Examples

```
>>> import numpy as np
>>> from mindspore import nn
>>> from mindspore.train import Model, LearningRateScheduler
>>> from mindspore import dataset as ds
...
>>> def learning_rate_function(lr, cur_step_num):
...     if cur_step_num%1000 == 0:
...         lr = lr*0.1
...     return lr
```

(continues on next page)

(continued from previous page)

```
...
>>> lr = 0.1
>>> momentum = 0.9
>>> net = nn.Dense(10, 5)
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), learning_rate=lr, momentum=momentum)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
...
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> model.train(1, dataset, callbacks=[LearningRateScheduler(learning_rate_function)],
...             dataset_sink_mode=False)
```

step_end(*run_context*)

Change the learning_rate at the end of step.

Parameters

run_context (RunContext) – Include some information of the model.

7.2.9 mindspore.train.LossMonitor

class mindspore.train.**LossMonitor**(*per_print_times=1*)

Monitor the loss in train or monitor the loss and eval metrics in fit.

If the loss is NAN or INF, it will terminate training.

Note: If per_print_times is 0, do not print loss.

Parameters

per_print_times (*int*) – How many steps to print once loss. During sink mode, it will print loss in the nearest step. Default: 1.

Raises

ValueError – If per_print_times is not an integer or less than zero.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, LossMonitor
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> loss_monitor = LossMonitor()
>>> model.train(10, dataset, callbacks=loss_monitor)
```

on_train_epoch_end(*run_context*)
When LossMonitor used in *mindspore.train.Model.fit()*, print eval metrics at the end of epoch if current epoch should do evaluation.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

step_end(*run_context*)
Print training loss at the end of step.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

7.2.10 mindspore.train.TrainFaultTolerance

class mindspore.train.**TrainFaultTolerance**(*ckpt_save_path=None*, ***kwargs*)
This callback is used to enable the TFT feature MindIO TFT and will execute TFT operations during training process, such as TFT init, report and exception handle.

Note: Required for Ascend graph mode only. And sink size must be less than or equal to 1.

Parameters

- **ckpt_save_path** (*str*) –Checkpoint save directory when failure occurs. When saved, a new directory named 'ttp_saved_checkpoints-step_{cur_step_num}' is created in that directory. Default: `None`.
- **kwargs** (*dict*) –Other dictionary type parameters.

Raises

- **Exception** –TFT init failed.
- **ModuleNotFoundError** –Mindio TFT whl package is not installed.

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

It's recommended to use the msrun startup method. Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import numpy as np
>>> import os
>>> import math
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import nn, ops, Parameter, train
>>> from mindspore.communication import init, get_rank
>>> from mindspore.common.initializer import initializer, HeUniform
>>> from mindspore.train import Model, TrainFaultTolerance
```

(continues on next page)

(continued from previous page)

```
>>> from mindspore import dataset as ds
>>> ms.set_context(mode=ms.GRAPH_MODE, jit_level='O2')
>>> ms.set_auto_parallel_context(parallel_mode=ms.ParallelMode.SEMI_AUTO_PARALLEL, pipeline_
 ↪stages=2)
>>> init()
>>> ms.set_seed(1)
>>> ms.set_auto_parallel_context(strategy_ckpt_config={"save_file":
...                              "./src_pipeline_strategys/src_strategy_{}.ckpt".format(get_
 ↪rank())})
>>> class MatMulCell(nn.Cell):
...     def __init__(self, param=None, shape=None):
...         super().__init__()
...         if shape is None:
...             shape = [28 * 28, 512]
...         weight_init = HeUniform(math.sqrt(5))
...         self.param = Parameter(initializer(weight_init, shape), name="param")
...         if param is not None:
...             self.param = param
...         self.print = ops.Print()
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         out = self.matmul(x, self.param)
...         self.print("out is:", out)
...         return out
>>>
>>> class Network(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.flatten = nn.Flatten()
...         self.layer1 = MatMulCell()
...         self.relu1 = nn.ReLU()
...         self.layer2 = nn.Dense(512, 512)
...         self.relu2 = nn.ReLU()
...         self.layer3 = nn.Dense(512, 10)
...
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.layer1(x)
...         x = self.relu1(x)
...         x = self.layer2(x)
...         x = self.relu2(x)
...         logits = self.layer3(x)
...         return logits
>>>
>>> net = Network()
>>> net.layer1.pipeline_stage = 0
>>> net.relu1.pipeline_stage = 0
>>> net.layer2.pipeline_stage = 0
>>> net.relu2.pipeline_stage = 1
>>> net.layer3.pipeline_stage = 1
>>>
>>> def create_dataset(batch_size):
...     dataset_path = os.getenv("DATA_PATH")
...     dataset = ds.MnistDataset(dataset_path)
...     image_transforms = [
...         ds.vision.Rescale(1.0 / 255.0, 0),
```

(continues on next page)

(continued from previous page)

```
...         ds.vision.Normalize(mean=(0.1307,), std=(0.3081,)),
...         ds.vision.HWC2CHW()
...     ]
...     label_transform = ds.transforms.TypeCast(ms.int32)
...     dataset = dataset.map(image_transforms, 'image')
...     dataset = dataset.map(label_transform, 'label')
...     dataset = dataset.batch(batch_size)
...     return dataset
>>>
>>> dataset = create_dataset(32)
>>>
>>> optimizer = nn.SGD(net.trainable_params(), 1e-2)
>>> optimizer_wrapper = nn.OptTFTWrapper(optimizer)
>>> loss_fn = nn.CrossEntropyLoss()
>>>
>>> net_with_loss = nn.Pipeline(nn.WithLossCell(net, loss_fn), 4)
>>> net_with_loss.set_train()
>>> model = Model(net_with_loss, optimizer=optimizer_wrapper)
>>> tft_cb = TrainFaultTolerance()
>>> loss_cb = train.LossMonitor(1)
>>> model.train(1, dataset, callbacks=[tft_cb, loss_cb])
```

end(*run_context*)

Unregister MindIO TFT on train end.

Parameters

run_context (RunContext) –Context of the train running. Refer to *mindspore.train.RunContext* for detail.

classmethod get_optimizer_wrapper(*origin_opt_cls*)

Optimizer wrapper func when using tft.

Parameters

origin_opt_cls (*Class*) –origin optimizer class.

on_train_begin(*run_context*)

Register train params to MindIO TFT on train beginning.

Parameters

run_context (RunContext) –Context of the train running. Refer to *mindspore.train.RunContext* for detail.

on_train_step_end(*run_context*)

Report status to MindIO TFT after every step finished.

Parameters

run_context (RunContext) –Context of the train running. Refer to *mindspore.train.RunContext* for detail.

7.2.11 mindspore.train.ModelCheckpoint

class mindspore.train.**ModelCheckpoint**(*prefix='CKP'*, *directory=None*, *config=None*)

The checkpoint callback class.

It is called to combine with train process and save the model and network parameters after training.

Note: In the distributed training scenario, please specify different directories for each training process to save the checkpoint file. Otherwise, the training may fail. If this callback is used in the Model function, the checkpoint file will saved parameters of the optimizer by default.

Parameters

- **prefix** (*Union[str, callable object]*) – The prefix name or callable object to generate name of checkpoint files. Default: `'CKP'` .
- **directory** (*Union[str, callable object]*) – The folder path where the checkpoint is stored, or the callable object used to generate the path. By default, the file is saved in the current directory. Default: `None` .
- **config** (CheckpointConfig) – Checkpoint strategy configuration. Default: `None` .

Raises

- **ValueError** – If *prefix* is not str or contains the '/' character and is not a callable object.
- **ValueError** – If *directory* is not str and is not a callable object.
- **TypeError** – If the config is not CheckpointConfig type.

Examples

```
>>> import numpy as np
>>> import mindspore.dataset as ds
>>> from mindspore import nn
>>> from mindspore.train import Model, ModelCheckpoint
>>>
>>> data = {"x": np.float32(np.random.rand(64, 10)), "y": np.random.randint(0, 5, (64,))}
>>> train_dataset = ds.NumpySlicesDataset(data=data).batch(32)
>>> net = nn.Dense(10, 5)
>>> crit = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> opt = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> ckpt_callback = ModelCheckpoint(prefix="myckpt")
>>> model = Model(network=net, optimizer=opt, loss_fn=crit)
>>> model.train(2, train_dataset, callbacks=[ckpt_callback])
```

end(*run_context*)

Save the last checkpoint after training finished.

Parameters

run_context (RunContext) – Context of the train running.

property latest_ckpt_file_name

Return the latest checkpoint path and file name.

step_end(*run_context*)

Save the checkpoint at the end of step.

Parameters

run_context (RunContext) –Context of the train running.

7.2.12 mindspore.train.OnRequestExit

class mindspore.train.**OnRequestExit**(*save_ckpt=True*, *save_mindir=True*, *file_name='Net'*, *directory='./'*, *config_file=None*, *sig=signal.SIGTERM*)

Respond to the user's closing request, exit the training or eval process, and save the checkpoint and mindir.

Register OnRequestExit Callback before training, when the user want to exit the training process and save the training data, could send the registered exit signal 'sig' to the training process or modify the 'GracefulExit' that a key in the JSON file specified by the 'config_file' to '1'. After the training process executes the current step, saves the current training status, including checkpoint and mindir, and then exit the training process.

Parameters

- **save_ckpt** (*bool*) –Whether save the checkpoint before the training process exit. Default: `True` .
- **save_mindir** (*bool*) –Whether save the mindir before the training process exit. Default: `True` .
- **file_name** (*str*) –The saved checkpoint and mindir file name, the checkpoint file add suffix '.ckpt', the mindir file add suffix '.mindir'. Default: `'Net'` .
- **directory** (*str*) –The path to save files. It will generate a 'rank_{id}' path by rank_id to save checkpoint and mindir. Default: `'./'` .
- **sig** (*int*) –The user registered exit signal, it must be a captureable and negligible signal. When the process receives the signal, exits the training or eval process. Default: `signal.SIGTERM` .
- **config_file** (*str*) –A json config file used to exit training process gracefully. Key: `{"GracefulExit": 1}` . Default: `None` .

Raises

- **ValueError** –If the 'save_ckpt' is not a bool.
- **ValueError** –If the 'save_mindir' is not a bool.
- **ValueError** –If the 'file_name' is not a str.
- **ValueError** –If the 'directory' is not a str.
- **ValueError** –If the 'sig' is not an int or the 'sig' is `signal.SIGTERM`.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, TimeMonitor
>>> import mindspore as ms
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> on_request_exit = ms.train.OnRequestExit(file_name='LeNet5')
>>> model.train(10, dataset, callbacks=on_request_exit)
```

on_eval_begin(*run_context*)
When the eval begin, register the handler for exit signal transferred by user.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_eval_end(*run_context*)
When the eval end, if received the exit signal, the checkpoint and mindir would be saved according to the user config.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_eval_step_end(*run_context*)
When the eval step end, if received the exit signal, set attribute '_stop_requested' of the 'run_context' to True. Then exit the eval process after this step eval.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_begin(*run_context*)
When the train begin, register the handler for exit signal transferred by user.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_end(*run_context*)
When the train end, if received the exit signal, the checkpoint and mindir would be saved according to the user config.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_epoch_end(*run_context*)
When the train epoch end, if received the exit signal, set the 'run_context' attribute '_stop_requested' to True. Then exit the training process after this epoch training.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_step_begin(*run_context*)
Check whether received the exit signal or whether the value of 'GracefulExit' in 'config_file' was changed to '1'.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_step_end(*run_context*)
Save checkpoint file or mindir file according to config, and exit the training process.

Parameters

run_context (RunContext) –Include some information of the model. For more details, please refer to *mindspore.train.RunContext*.

7.2.13 mindspore.train.ReduceLROnPlateau

class mindspore.train.**ReduceLROnPlateau**(*monitor='eval_loss'*, *factor=0.1*, *patience=10*, *verbose=False*, *mode='auto'*, *min_delta=1e-4*, *cooldown=0*, *min_lr=0*)

Reduce learning rate when the monitor has stopped improving.

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors the training process and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Note: Learning rate grouping is not supported now.

Parameters

- **monitor** (*str*) – quantity to be monitored. If evaluation is performed on the end of train epochs, the valid monitors can be `"loss"`, `"eval_loss"` or metric names passed when instantiate the *Model*; otherwise the valid monitor is `"loss"`. When *monitor* is `"loss"`, if train network has multiple outputs, the first element will be returned as training loss. Default: `'eval_loss'`.
- **factor** (*float*) – factor by which the learning rate will be reduced. *new_lr = lr * factor*. Default: `0.1` .
- **patience** (*int*) – *monitor* value is better than history best value over *min_delta* is seen as improvement, *patience* is number of epochs with no improvement that would be waited. When the waiting counter *self.wait* is larger than or equal to *patience*, the lr will be reduced. Default: `10` .
- **verbose** (*bool*) – If False: quiet, if True: print related information. Default: `False` .
- **mode** (*str*) – one of *{'auto', 'min', 'max'}*. Default: `'auto'` .
  - In `'min'` mode, the learning rate will be reduced when the quantity monitored has stopped decreasing.
  - In `'max'` mode it will be reduced when the quantity monitored has stopped increasing.
  - In `'auto'` mode, the direction is automatically inferred from the name of the monitored quantity.
- **min_delta** (*float*) – threshold for measuring the new optimum, to only focus on significant changes. Default: `1e-4` .
- **cooldown** (*int*) – number of epochs to wait before resuming normal operation after lr has been reduced. Default: `0` .
- **min_lr** (*float*) – lower bound on the learning rate. Default: `0` .

Raises

- **ValueError** – *mode* not in `'auto'`, `'min'` or `'max'`.
- **ValueError** – The monitor value is not a scalar.
- **ValueError** – The learning rate is not a Parameter.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, ReduceLROnPlateau
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics={"acc"})
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> cb = ReduceLROnPlateau(monitor="acc", patience=3, verbose=True)
>>> model.fit(10, dataset, callbacks=cb)
```

on_train_begin(*run_context*)

Initialize variables at the begin of training.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

on_train_epoch_end(*run_context*)

monitors the training process and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Parameters

run_context (RunContext) –Context information of the model. For more details, please refer to *mindspore.train.RunContext*.

7.2.14 mindspore.train.RunContext

class mindspore.train.**RunContext**(*original_args*)

Hold and manage information about the model.

RunContext is mainly used to collect context-related information about the model during training or eval and pass it into the Callback object as an input parameter to share information.

Callback objects not only can obtain the Model context information by calling by *RunContext.original_args()* and add extra attributes to the information, but also can stop the training process by calling *request_stop* method.

RunContext.original_args() holds the model context information as a dictionary variable, and different attributes of the dictionary are stored in training or eval process. Details are as follows:

Attributes supported in train	Attributes supported in eval	meaning
train_network		train network with optimizer and loss
epoch_num		Number of train epochs
train_dataset		the train dataset
loss_fn		the loss function
optimizer		the optimizer
parallel_mode		the parallel mode
device_number		the device number
train_dataset_element		the train data element of current step
last_save_ckpt_step		the last step num of save ckpt
latest_ckpt_file		the ckpt file
cur_epoch_num		number of current epoch
	eval_network	the evaluate network
	valid_dataset	the valid dataset
	metrics	the evaluate metrics
mode	mode	"train" or "eval"
batch_num	batch_num	the train/eval batch number
list_callback	list_callback	callback list
network	network	basic network
cur_step_num	cur_step_num	the train/eval step number
dataset_sink_mode	dataset_sink_mode	the train/eval sink mode
net_outputs	net_outputs	network output results

Parameters

original_args (*dict*) – Holding the related information of model.

Examples

```
>>> from mindspore import Tensor
>>> from mindspore.train import RunContext
>>> cb_params = {}
>>> cb_params["cur_epoch_num"] = 4
>>> cb_params["epoch_num"] = 4
>>> cb_params["cur_step_num"] = 2
>>> cb_params["batch_num"] = 2
>>> cb_params["net_outputs"] = Tensor(2.0)
>>> run_context = RunContext(cb_params)
>>> whether_stop = run_context.get_stop_requested()
```

get_stop_requested()

Return whether a stop is requested or not.

Returns

bool, if true, model.train() stops iterations.

original_args()

Get the _original_args object.

Returns

Dict, an object that holds the original arguments of model.

request_stop()
Set stop requirement during training or eval.

Callbacks can use this function to request stop of iterations. model.train() checks whether this is called or not.

7.2.15 mindspore.train.TimeMonitor

class mindspore.train.**TimeMonitor**(*data_size=None*, *data_time=False*)
Monitor the time in train or eval process.

Parameters

- **data_size** (*int*) –How many steps are the intervals between print information each time. if the program get *batch_num* during training, *data_size* will be set to *batch_num*, otherwise *data_size* will be used. If the program does not get *batch_num* during training, meanwhile *data_size* does not set, the program will report an error. Default: `None` .
- **data_time** (*bool*) –Whether to show the average time of fetching data in Host. Note that data fetch and network compute are processed sequentially in non dataset sink mode, while they are asynchronous in dataset sink mode. Default: `False` .

Raises

- **ValueError** –If data_size is not positive int.
- **TypeError** –If data_time is not bool.

Examples

```
>>> from mindspore import nn
>>> from mindspore.train import Model, TimeMonitor
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
>>> optim = nn.Momentum(net.trainable_params(), 0.01, 0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> time_monitor = TimeMonitor()
>>> model.train(10, dataset, callbacks=time_monitor)
```

epoch_begin(*run_context*)
Record time at the beginning of epoch.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

epoch_end(*run_context*)
Print process cost time at the end of epoch.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

on_train_step_begin(*run_context*)
Record time at the beginning of step.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

on_train_step_end(*run_context*)
Record time at the end of step.

Parameters

run_context (RunContext) –Context of the process running. For more details, please refer to *mindspore.train.RunContext*.

7.3 Evaluation Metrics

API Name	Description	Supported Platforms
mindspore.train.Accuracy	Calculates the accuracy for classification and multilabel data.	Ascend GPU CPU
mindspore.train.BleuScore	Calculates the BLEU score.	Ascend GPU CPU
mindspore.train.ConfusionMatrix	Computes the Confusion Matrix, which is commonly used to evaluate the performance of classification models, including binary classification and multiple classification.	Ascend GPU CPU
mindspore.train.ConfusionMatrixMetric	Computes metrics related to confusion matrix.	Ascend GPU CPU
mindspore.train.CosineSimilarity	Computes representation similarity.	Ascend GPU CPU
mindspore.train.Dice	The Dice coefficient is a set similarity metric.	Ascend GPU CPU
mindspore.train.F1	Calculates the F1 score.	Ascend GPU CPU
mindspore.train.Fbeta	Calculates the Fbeta score.	Ascend GPU CPU
mindspore.train.HausdorffDistance	Calculates the Hausdorff distance.	Ascend GPU CPU
mindspore.train.Loss	Calculates the average of the loss.	Ascend GPU CPU
mindspore.train.MAE	Calculates the mean absolute error(MAE).	Ascend GPU CPU
mindspore.train.MeanSurfaceDistance	Computes the Average Surface Distance from *y_pred* to *y* under the default setting.	Ascend GPU CPU
mindspore.train.Metric	Base class of metric, which is used to evaluate metrics.	Ascend GPU CPU
mindspore.train.MSE	Measures the mean squared error(MSE).	Ascend GPU CPU
mindspore.train.OcclusionSensitivity	Calculates the occlusion sensitivity of the model for a given image, which illustrates which parts of an image are most important for a network's classification.	Ascend GPU CPU
mindspore.train.Perplexity	Computes perplexity.	Ascend GPU CPU
mindspore.train.Precision	Calculates precision for classification and multilabel data.	Ascend GPU CPU
mindspore.train.Recall	Calculates recall for classification and multilabel data.	Ascend GPU CPU
mindspore.train.ROC	Calculates the ROC curve.	Ascend GPU CPU
mindspore.train.RootMeanSquareDistance	Computes the Root Mean Square Surface Distance from *y_pred* to *y* under the default setting.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.train.Top1CategoricalAccuracy	Calculates the top-1 categorical accuracy.	Ascend GPU CPU
mindspore.train.Top5CategoricalAccuracy	Calculates the top-5 categorical accuracy.	Ascend GPU CPU
mindspore.train.TopKCategoricalAccuracy	Calculates the top-k categorical accuracy.	Ascend GPU CPU

7.3.1 mindspore.train.Accuracy

class mindspore.train.**Accuracy**(*eval_type='classification'*)

Calculates the accuracy for classification and multilabel data.

The accuracy class creates two local variables, the correct number and the total number that are used to compute the frequency with which y_pred matches y. This frequency is the accuracy.

$$\text{accuracy} = \frac{\text{true\_positive} + \text{true\_negative}}{\text{true\_positive} + \text{true\_negative} + \text{false\_positive} + \text{false\_negative}}$$

Parameters

eval_type (*str*) – The metric to calculate the accuracy over a dataset. Supports `'classification'` and `'multilabel'`. `'classification'` means the dataset label is single. `'multilabel'` means the dataset has multiple labels. Default: `'classification'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import Accuracy
>>>
>>> y_pred = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]), mindspore.float32)
>>> y_true = Tensor(np.array([1, 0, 1]), mindspore.float32)
>>> metric = Accuracy('classification')
>>> metric.clear()
>>> metric.update(y_pred, y_true)
>>> accuracy = metric.eval()
>>> print(accuracy)
0.6666666666666666
```

clear()

Clears the internal evaluation result.

eval()

Computes the accuracy.

Returns

np.float64, the computed result.

Raises

RuntimeError –If the sample size is 0.

update(**inputs*)

Updates the local variables. For `'classification'`, if the index of the maximum of the predict value matches the label, the predict result is correct. For `'multilabel'`, the predict value match the label, the predict result is correct.

Parameters

inputs –Logits and labels. *y_pred* stands for logits, *y* stands for labels. *y_pred* and *y* must be a *Tensor*, a list or an array.

- For the `'classification'` evaluation type, *y_pred* is a list of floating numbers in range $[0, 1]$ and the shape is (N, C) in most cases (not strictly), where N is the number of cases and C is the number of categories. *y* must be in one-hot format that shape is (N, C), or can be transformed to one-hot format that shape is $(N,)$.
- For `'multilabel'` evaluation type, the value of *y_pred* and *y* can only be 0 or 1, indices with 1 indicate the positive category. The shape of *y_pred* and *y* are both (N, C).

Raises

- **ValueError** –If the number of the inputs is not 2.
- **ValueError** –class numbers of last input predicted data and current predicted data not match.

7.3.2 mindspore.train.BleuScore

class mindspore.train.**BleuScore**(*n_gram=4*, *smooth=False*)

Calculates the BLEU score. BLEU (bilingual evaluation understudy) is a metric for evaluating the quality of text translated by machine.

Parameters

- **n_gram** (*int*) –The n_gram value ranges from 1 to 4. Default: `4` .
- **smooth** (*bool*) –Whether or not to apply smoothing. Default: `False` .

Raises

ValueError –If the value range of n_gram is not from 1 to 4.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> from mindspore.train import BleuScore
>>>
>>> candidate_corpus = [['i', 'have', 'a', 'pen', 'on', 'my', 'desk']]
>>> reference_corpus = [[['i', 'have', 'a', 'pen', 'in', 'my', 'desk'],
...                      ['there', 'is', 'a', 'pen', 'on', 'the', 'desk']]]
>>> metric = BleuScore()
>>> metric.clear()
>>> metric.update(candidate_corpus, reference_corpus)
>>> bleu_score = metric.eval()
>>> print(bleu_score)
0.5946035575013605
```

clear()

Clear the internal evaluation result.

eval()
Computes the bleu score.

Returns

numpy.float64, the bleu score.

Raises

RuntimeError –If the update method is not called first, an error will be reported.

update(**inputs*)
Updates the internal evaluation result with *candidate_corpus* and *reference_corpus*.

Parameters

inputs (*iterator*) –Input *candidate_corpus* and *reference_corpus*. *candidate_corpus* and *reference_corpus* are both a list. The *candidate_corpus* is an iterable of machine translated corpus. The *reference_corpus* is an iterable object of iterables of reference corpus.

Raises

- **ValueError** –If the number of inputs is not 2.
- **ValueError** –If the lengths of *candidate_corpus* and *reference_corpus* are not equal.

7.3.3 mindspore.train.ConfusionMatrix

class mindspore.train.**ConfusionMatrix**(*num_classes*, *normalize='no_norm'*, *threshold=0.5*)
Computes the Confusion Matrix, which is commonly used to evaluate the performance of classification models, including binary classification and multiple classification.

If you only need Confusion Matrix, use this class. If you want to calculate other metrics, such as 'PPV', 'TPR', 'TNR', etc., use class *mindspore.train.ConfusionMatrixMetric*.

Parameters

- **num_classes** (*int*) –Number of classes in the dataset.
- **normalize** (*str*) –Normalization mode for Confusion Matrix. Default: `"no_norm"`. Choose from:
  - `"no_norm"`: No Normalization is used. Default: `None`.
  - `"target"`: Normalization based on target value.
  - `"prediction"`: Normalization based on predicted value.
  - `"all"`: Normalization over the whole matrix.
- **threshold** (*float*) –The threshold used to compare with the input tensor. Default: `0.5`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import ConfusionMatrix
>>>
>>> x = Tensor(np.array([1, 0, 1, 0]))
>>> y = Tensor(np.array([1, 0, 0, 1]))
>>> metric = ConfusionMatrix(num_classes=2, normalize='no_norm', threshold=0.5)
>>> metric.clear()
>>> metric.update(x, y)
>>> output = metric.eval()
>>> print(output)
[[1. 1.]
 [1. 1.]]
```

clear()
Clears the internal evaluation result.

eval()
Computes confusion matrix.

Returns

numpy.ndarray, the computed result.

update(**inputs*)
Update state with *y_pred* and *y*.

Parameters

inputs (*tuple*) −Input *y_pred* and *y*. *y_pred* and *y* are a *Tensor*, list or numpy.ndarray. *y_pred* is the predicted value, *y* is the true value. The shape of *y_pred* is $(N, C, ...)$ or $(N, ...)$. The shape of *y* is $(N, ...)$.

Raises

- **ValueError** −If the number of inputs is not 2.
- **ValueError** −If the dims of *y_pred* and *y* are not equal.

7.3.4 mindspore.train.ConfusionMatrixMetric

class mindspore.train.**ConfusionMatrixMetric**(*skip_channel=True*, *metric_name='sensitivity'*, *calculation_method=False*, *decrease='mean'*)
Computes metrics related to confusion matrix. The calculation based on full-scale tensor, average values of batch, class channel and iteration are collected. All metrics supported by the interface are listed in comments of *metric_name*.

- If you want to calculate metrics related to confusion matrix, such as 'PPV', 'TPR', 'TNR', use this class.
- If you only want to calculate confusion matrix, please use *mindspore.train.ConfusionMatrix*.

Parameters

- **skip_channel** (*bool*) −Whether to skip the measurement calculation on the first channel of the predicted output. Default: `True`.
- **metric_name** (*str*) −Names of supported metrics , users can also set the industry common aliases for them. Choose from: ["sensitivity", "specificity", "precision", "negative predictive value", "miss rate", "fall out", "false discovery rate", "false omission rate", "prevalence threshold", "threat score", "accuracy", "balanced accuracy", "f1 score", "matthews correlation coefficient", "fowlkes mallows index", "informedness", "markedness"]. Default: `"sensitivity"`.

- **calculation_method** (*bool*) –If `True`, the measurement for each sample will be calculated first. If not, the confusion matrix of all samples will be accumulated first. As for classification task, 'calculation_method' should be `False`. Default: `False` .
- **decrease** (*str*) –The reduction method on data batch. *decrease* takes effect only when calculation_method is True. Default: `"mean"` . Choose from: ["none", "mean", "sum", "mean_batch", "sum_batch", "mean_channel", "sum_channel"].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import ConfusionMatrixMetric
>>>
>>> metric = ConfusionMatrixMetric(skip_channel=True, metric_name="tpr",
...                                calculation_method=False, decrease="mean")
>>> metric.clear()
>>> x = Tensor(np.array([[[0], [1]], [[1], [0]]]))
>>> y = Tensor(np.array([[[0], [1]], [[0], [1]]]))
>>> metric.update(x, y)
>>> avg_output = metric.eval()
>>> print(avg_output)
[0.5]
```

clear()
Clears the internal evaluation result.

eval()
Computes confusion matrix metric.

Returns

ndarray, the computed result.

update(**inputs*)
Update state with predictions and targets.

Parameters

inputs (*tuple*) –Input *y_pred* and *y*. *y_pred* and *y* are a *Tensor*, list or numpy.ndarray.

- y_pred (ndarray): The batch data shape is $(N, C, ...)$ or $(N, ...)$, representing onehot format or category index format respectively. As for classification tasks, y_pred should have the shape (B, N) where N is larger than 1. As for segmentation tasks, the shape should be (B, N, H, W) or (B, N, H, W, D).
- y (ndarray): It must be one-hot format. The batch data shape is $(N, C, ...)$.

Raises

ValueError –If the number of the inputs is not 2.

7.3.5 mindspore.train.CosineSimilarity

class mindspore.train.**CosineSimilarity**(*similarity='cosine'*, *reduction='none'*, *zero_diagonal=True*)

Computes representation similarity.

Parameters

- **similarity** (*str*) – the computation logit. `'cosine'` means computing similarity. `'dot'` means computing dots of arrays, Default: `'cosine'` .
- **reduction** (*str*) – Specifies the reduction to be applied to the output. Support `'none'`, `'sum'`, `'mean'` (all along dim -1). Default: `'none'` .
- **zero_diagonal** (*bool*) – If `True`, diagonals of results will be set to zero. Default: `True` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore.train import CosineSimilarity
>>>
>>> test_data = np.array([[1, 3, 4, 7], [2, 4, 2, 5], [3, 1, 5, 8]])
>>> metric = CosineSimilarity()
>>> metric.clear()
>>> metric.update(test_data)
>>> square_matrix = metric.eval()
>>> print(square_matrix)
[[0.  0.94025615  0.95162452]
 [0.94025615  0.  0.86146098]
 [0.95162452  0.86146098  0.]]
```

clear()

Clears the internal evaluation result.

eval()

Computes the similarity matrix.

Returns

numpy.ndarray. The similarity matrix.

Raises

RuntimeError – If the update method is not called first, an error will be reported.

update(*inputs*)

Updates the internal evaluation result with 'inputs'.

Parameters

inputs (*Union[Tensor, list, numpy.ndarray]*) – The input matrix including *y* and *y_pred*.

7.3.6 mindspore.train.Dice

class mindspore.train.**Dice**(*smooth=1e-5*)

The Dice coefficient is a set similarity metric. It is used to calculate the similarity between two samples. The value of the Dice coefficient is 1 when the segmentation result is the best and is 0 when the segmentation result is the worst. The Dice coefficient indicates the ratio of the area between two objects to the total area. The function is shown as follows:

$$dice = \frac{2 * (pred \bigcap true)}{pred \bigcup true}$$

Parameters

smooth (*float*) – A term added to the denominator to improve numerical stability. Should be greater than 0. Default: `1e-5`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import Dice
>>>
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([[0, 1], [1, 0], [0, 1]]))
>>> metric = Dice(smooth=1e-5)
>>> metric.clear()
>>> metric.update(x, y)
>>> dice = metric.eval()
>>> print(dice)
0.20467791371802546
```

clear()

Clears the internal evaluation result.

eval()

Computes the Dice.

Returns

Float, the computed result.

Raises

RuntimeError – If the total number of samples is 0.

update(**inputs*)

Updates the internal evaluation result *y_pred* and *y*.

Parameters

inputs (*tuple*) – Input *y_pred* and *y*. *y_pred* and *y* are Tensor, list or numpy.ndarray. *y_pred* is the predicted value, *y* is the true value. The shape of *y_pred* and *y* are both $(N, ...)$.

Raises

- **ValueError** – If the number of the inputs is not 2.
- **ValueError** – If y_pred and y do not have the same shape.

7.3.7 mindspore.train.F1

class mindspore.train.**F1**

Calculates the F1 score. F1 is a special case of Fbeta when beta is 1. Refer to class *mindspore.train.Fbeta* for more details.

$$F_1 = \frac{2 \cdot true\_positive}{2 \cdot true\_positive + false\_negative + false\_positive}$$

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import F1
>>>
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([1, 0, 1]))
>>> metric = F1()
>>> metric.update(x, y)
>>> result = metric.eval()
>>> print(result)
[0.66666667 0.66666667]
```

7.3.8 mindspore.train.Fbeta

class mindspore.train.**Fbeta**(*beta*)

Calculates the Fbeta score.

Fbeta score is a weighted mean of precision and recall.

$$F_\beta = \frac{(1+\beta^2) \cdot true\_positive}{(1+\beta^2) \cdot true\_positive + \beta^2 \cdot false\_negative + false\_positive}$$

Parameters

beta (*Union[float, int]*) – Beta coefficient in the F measure. *beta* should be greater than 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import Fbeta
>>>
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([1, 0, 1]))
>>> metric = Fbeta(1)
```

(continues on next page)

(continued from previous page)

```
>>> metric.clear()
>>> metric.update(x, y)
>>> fbeta = metric.eval()
>>> print(fbeta)
[0.66666667 0.66666667]
```

clear()
Clears the internal evaluation result.

eval(*average=False*)
Computes the fbeta.

Parameters

average (*bool*) −Whether to calculate the average fbeta. Default: `False`.

Returns

numpy.ndarray or numpy.float64, the computed result.

update(**inputs*)
Updates the internal evaluation result *y_pred* and *y*.

Parameters

inputs −Input *y_pred* and *y*. *y_pred* and *y* are Tensor, list or numpy.ndarray. *y_pred* is in most cases (not strictly) a list of floating numbers in range $[0, 1]$ and the shape is (N, C), where N is the number of cases and C is the number of categories. y contains values of integers. The shape is (N, C) if one-hot encoding is used. Shape can also be $(N,)$ if category index is used.

Raises

- **ValueError** −class numbers of last input predicted data and current predicted data not match.
- **ValueError** −If the predicted value and true value contain different classes.

7.3.9 mindspore.train.HausdorffDistance

class mindspore.train.**HausdorffDistance**(*distance_metric='euclidean'*, *percentile=None*, *directed=False*, *crop=True*)

Calculates the Hausdorff distance. Hausdorff distance is the maximum and minimum distance between two point sets. Given two feature sets A and B, the Hausdorff distance between two point sets A and B is defined as follows:

$$
\begin{aligned}
H(A, B) &= \max[h(A, B), h(B, A)] \\
h(A, B) &= \max_{a \in A}\{\min_{b \in B}\|a - b\|\} \\
h(B, A) &= \max_{b \in B}\{\min_{a \in A}\|b - a\|\}
\end{aligned}
$$

where $h(A, B)$ is the maximum distance of a set A to the nearest point in the set B, $h(B, A)$ is the maximum distance of a set B to the nearest point in the set A. The distance calculation is oriented, which means that most of times $h(A, B)$ is not equal to $h(B, A)$. $H(A, B)$ is the two-way Hausdorff distance.

Parameters

- **distance_metric** (*str*) −Three distance measurement methods are supported: `"euclidean"` (Euclidean Distance) , `"chessboard"` (Chessboard Distance, Chebyshev Distance) or `"taxicab"` (Taxicab Distance, Manhattan Distance). Default: `"euclidean"` .

- **percentile** (*float*) –Floating point numbers between 0 and 100. Specify the percentile parameter to get the percentile of the Hausdorff distance. Default: `None` .
- **directed** (*bool*) –If True, it only calculates h(y_pred, y) distance, otherwise, max(h(y_pred, y), h(y, y_pred)) will be returned. Default: `False` .
- **crop** (*bool*) –Crop input images and only keep the foregrounds. In order to maintain two inputs' shapes, here the bounding box is achieved by (y_pred | y) which represents the union set of two images. Default: `True` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import HausdorffDistance
>>>
>>> x = Tensor(np.array([[3, 0, 1], [1, 3, 0], [1, 0, 2]]))
>>> y = Tensor(np.array([[0, 2, 1], [1, 2, 1], [0, 0, 1]]))
>>> metric = HausdorffDistance()
>>> metric.clear()
>>> metric.update(x, y, 0)
>>> mean_average_distance = metric.eval()
>>> print(mean_average_distance)
1.4142135623730951
```

clear()
Clears the internal evaluation result.

eval()
Calculate the no-directed or directed Hausdorff distance.

Returns

numpy.float64, the hausdorff distance.

Raises

RuntimeError –If the update method is not called first, an error will be reported.

update(**inputs*)
Updates the internal evaluation result with the inputs: 'y_pred', 'y' and 'label_idx'.

Parameters

inputs –Input 'y_pred', 'y' and 'label_idx'. 'y_pred' and 'y' are a *Tensor*, list or numpy.ndarray. 'y_pred' is the predicted binary image. 'y' is the actual binary image. Data type of 'label_idx' is int or float.

Raises

- **ValueError** –If the number of the inputs is not 3.
- **TypeError** –If the data type of *label_idx* is not int or float.
- **ValueError** –If the value of *label_idx* is not in *y_pred* or *y*.
- **ValueError** –If *y_pred* and *y* have different shapes.

7.3.10 mindspore.train.Loss

class mindspore.train.**Loss**

Calculates the average of the loss. If method 'update' is called every n iterations, the result of evaluation will be:

$$loss = \frac{\sum_{k=1}^{n} loss_k}{n}$$

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import Loss
>>>
>>> x = Tensor(np.array(0.2), mindspore.float32)
>>> loss = Loss()
>>> loss.clear()
>>> loss.update(x)
>>> result = loss.eval()
>>> print(result)
0.20000000298023224
```

clear()

Clears the internal evaluation result.

eval()

Calculates the average of the loss.

Returns

numpy.float64. The average of the loss.

Raises

RuntimeError – If the total number is 0.

update(**inputs*)

Updates the internal evaluation result.

Parameters

inputs – Inputs contain only one element, the element is loss. The dimension of loss must be 0 or 1.

Raises

- **ValueError** – If the length of inputs is not 1.
- **ValueError** – If the dimension of loss is not 1 or 0.

7.3.11 mindspore.train.MAE

class mindspore.train.**MAE**

Calculates the mean absolute error(MAE).

Creates a criterion that measures the MAE between each element in the input: y_pred and the target: y.

$$\text{MAE} = \frac{\sum_{i=1}^{n} \|y\_pred_i - y_i\|}{n}$$

where n is batch size.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import MAE
>>>
>>> x = Tensor(np.array([0.1, 0.2, 0.6, 0.9]), mindspore.float32)
>>> y = Tensor(np.array([0.1, 0.25, 0.7, 0.9]), mindspore.float32)
>>> error = MAE()
>>> error.clear()
>>> error.update(x, y)
>>> result = error.eval()
>>> print(result)
0.037499990314245224
```

clear()

Clears the internal evaluation result.

eval()

Computes the mean absolute error(MAE).

Returns

numpy.float64. The computed result.

Raises

RuntimeError – If the total number of samples is 0.

update(**inputs*)

Updates the internal evaluation result y_{pred} and y.

Parameters

inputs – Input *y_pred* and *y* for calculating MAE where the shape of *y_pred* and *y* are both N-D and the shape should be the same.

Raises

ValueError – If the number of the input is not 2.

7.3.12 mindspore.train.MeanSurfaceDistance

class mindspore.train.**MeanSurfaceDistance**(*symmetric=False*, *distance_metric='euclidean'*)

Computes the Average Surface Distance from *y_pred* to *y* under the default setting. It measures how much, on average, the surface varies between the segmentation and the GT (ground truth).

Given two sets A and B, S(A) denotes the set of surface voxels of A, the shortest distance of an arbitrary voxel v to S(A) is defined as:

$$\text{dis}(v, S(A)) = \min_{s_A \in S(A)} \|v - s_A\|$$

The Average Surface Distance from set(B) to set(A) is given by:

$$AvgSurDis(B \to A) = \frac{\sum_{s_B \in S(B)} \text{dis}(s_B, S(A))}{|S(B)|}$$

Where the ||*|| denotes a distance measure. |*| denotes the number of elements.

The mean of surface distance from set(B) to set(A) and from set(A) to set(B) is:

$$MeanSurDis(A \leftrightarrow B) = \frac{\sum_{s_A \in S(A)} \text{dis}(s_A, S(B)) + \sum_{s_B \in S(B)} \text{dis}(s_B, S(A))}{|S(A)| + |S(B)|}$$

Parameters

- **distance_metric** (*str*) –Three measurement methods are supported: `"euclidean"` (Euclidean Distance) , `"chessboard"` (Chessboard Distance, Chebyshev Distance) or `"taxicab"` (Taxicab Distance, Manhattan Distance) Default: `"euclidean"` .
- **symmetric** (*bool*) –Whether to calculate the Mean Surface Distance between y_pred and y. If False, it only calculates $AvgSurDis(y\_pred \to y)$, otherwise, the mean of distance from *y_pred* to *y* and from *y* to *y_pred*, i.e. $MeanSurDis(y\_pred \leftrightarrow y)$, will be returned. Default: `False` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import MeanSurfaceDistance
>>> x = Tensor(np.array([[3, 0, 1], [1, 3, 0], [1, 0, 2]]))
>>> y = Tensor(np.array([[0, 2, 1], [1, 2, 1], [0, 0, 1]]))
>>> metric = MeanSurfaceDistance(symmetric=False, distance_metric="euclidean")
>>> metric.clear()
>>> metric.update(x, y, 0)
>>> mean_average_distance = metric.eval()
>>> print(mean_average_distance)
0.8047378541243649
```

clear()

Clears the internal evaluation result.

eval()

Calculate mean surface distance.

Returns

numpy.float64. The mean surface distance value.

Raises

RuntimeError –If the update method is not called first, an error will be reported.

update(**inputs*)

Updates the internal evaluation result 'y_pred', 'y' and 'label_idx'.

Parameters

inputs –Input 'y_pred', 'y' and 'label_idx'. 'y_pred' and 'y' are a Tensor, list or numpy.ndarray. 'y_pred' is the predicted binary image. 'y' is the actual binary image. 'label_idx', the data type of *label_idx* is int.

Raises

- **ValueError** –If the number of the inputs is not 3.
- **TypeError** –If the data type of label_idx is not int or float.
- **ValueError** –If the value of label_idx is not in *y_pred* or *y*.
- **ValueError** –If *y_pred* and *y* have different shapes.

7.3.13 mindspore.train.Metric

class mindspore.train.**Metric**

Base class of metric, which is used to evaluate metrics.

The *clear*, *update*, and *eval* should be called when evaluating metric, and they should be overridden by subclasse. *update* will accumulate intermediate results in the evaluation process, *eval* will evaluate the final result, and *clear* will reinitialize the intermediate results.

Never use this class directly, but instantiate one of its subclasses instead, for examples, *mindspore.train.MAE*, *mindspore.train.Recall* etc.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>>
>>> class MyMAE(ms.train.Metric):
...     def __init__(self):
...         super(MyMAE, self).__init__()
...         self.clear()
...
...     def clear(self):
...         self._abs_error_sum = 0
...         self._samples_num = 0
...
...     def update(self, *inputs):
...         y_pred = inputs[0].asnumpy()
...         y = inputs[1].asnumpy()
...         abs_error_sum = np.abs(y - y_pred)
...         self._abs_error_sum += abs_error_sum.sum()
```

(continues on next page)

(continued from previous page)

```
...         self._samples_num += y.shape[0]
...
...     def eval(self):
...         return self._abs_error_sum / self._samples_num
>>>
>>> x = ms.Tensor(np.array([[0.1, 0.2, 0.6, 0.9], [0.1, 0.2, 0.6, 0.9]]), ms.float32)
>>> y = ms.Tensor(np.array([[0.1, 0.1, 0.1, 0.1], [0.1, 0.1, 0.1, 0.1]]), ms.float32)
>>> y2 = ms.Tensor(np.array([[0.1, 0.25, 0.7, 0.9], [0.1, 0.25, 0.7, 0.9]]), ms.float32)
>>> metric = MyMAE().set_indexes([0, 2])
>>> metric.clear()
>>> # indexes is [0, 2], using x as logits, y2 as label.
>>> metric.update(x, y, y2)
>>> accuracy = metric.eval()
>>> print(accuracy)
1.399999976158142
>>> print(metric.indexes)
[0, 2]
```

abstract clear()

An interface describes the behavior of clearing the internal evaluation result.

Note: All subclasses must override this interface.

---

abstract eval()

An interface describes the behavior of computing the evaluation result.

Note: All subclasses must override this interface.

---

property indexes

Get the current indexes value. The default value is None and can be changed by *set_indexes*.

set_indexes(*indexes*)

This interface is to rearrange the inputs of *update*.

Given (label0, label1, logits), set the *indexes* to [2, 1] then the (logits, label1) will be the actually inputs of *update*.

Note: When customize a metric, decorate the *update* function with the decorator *mindspore.train.rearrange_inputs()* for the *indexes* to take effect.

Parameters

indexes (*List(int)*) – The order of logits and labels to be rearranged.

Outputs:

Metric, its original Class instance.

Raises

ValueError – If the type of input 'indexes' is not a list or its elements are not all int.

abstract update(**inputs*)

An interface describes the behavior of updating the internal evaluation result.

Note: All subclasses must override this interface.

Parameters

inputs –A variable-length input argument list, usually are the logits and the corresponding labels.

7.3.14 mindspore.train.MSE

class mindspore.train.**MSE**

Measures the mean squared error(MSE).

Creates a criterion that measures the MSE (squared L2 norm) between each element in the prediction and the ground truth: y_pred and: y.

$$\text{MSE}(y\_pred,\ y) = \frac{\sum_{i=1}^{n}(y\_pred_i - y_i)^2}{n}$$

where n is batch size.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import MSE
>>>
>>> x = Tensor(np.array([0.1, 0.2, 0.6, 0.9]), mindspore.float32)
>>> y = Tensor(np.array([0.1, 0.25, 0.5, 0.9]), mindspore.float32)
>>> error = MSE()
>>> error.clear()
>>> error.update(x, y)
>>> result = error.eval()
>>> print(result)
0.0031250009778887033
```

clear()

Clear the internal evaluation result.

eval()

Computes the mean squared error(MSE).

Returns

numpy.float64. The computed result.

Raises

RuntimeError –If the number of samples is 0.

update(**inputs*)

Updates the internal evaluation result y_{pred} and y.

Parameters

inputs – Input y_pred and y for calculating the MSE where the shape of y_pred and y are both N-D and the shape should be the same.

Raises

ValueError – If the number of inputs is not 2.

7.3.15 mindspore.train.OcclusionSensitivity

class mindspore.train.**OcclusionSensitivity**(*pad_val=0.0*, *margin=2*, *n_batch=128*, *b_box=None*)

Calculates the occlusion sensitivity of the model for a given image, which illustrates which parts of an image are most important for a network's classification.

Occlusion sensitivity refers to how the predicted probability changes with the change of the occluded part of an image. The higher the value in the output image is, the greater the certainty decline of the category after masking, indicating that the occluded area is more important in the decision-making process.

Parameters

- **pad_val** (*float*) – The padding value of the occluded part in an image. Default: `0.0` .
- **margin** (*Union[int, Sequence]*) – Create a cuboid / cube size of pixel points around the voxel you want to occlude. Default: `2` .
- **n_batch** (*int*) – number of images in a batch. Default: `128` .
- **b_box** (*Sequence*) – Bounding box on which to perform the analysis. The output image will also match in size. There should be a minimum and maximum for all dimensions except batch: `[min1, max1, min2, max2,...]`. If no bounding box is supplied, this will be the same size as the input image. If a bounding box is used, the output image will be cropped to this size. Default: `None` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import nn, Tensor
>>> from mindspore.train import OcclusionSensitivity
>>>
>>> class DenseNet(nn.Cell):
...     def __init__(self):
...         super(DenseNet, self).__init__()
...         w = np.array([[0.1, 0.8, 0.1, 0.1],[1, 1, 1, 1]]).astype(np.float32)
...         b = np.array([0.3, 0.6]).astype(np.float32)
...         self.dense = nn.Dense(4, 2, weight_init=Tensor(w), bias_init=Tensor(b))
...
...     def construct(self, x):
...         return self.dense(x)
>>>
>>> model = DenseNet()
>>> test_data = np.array([[0.1, 0.2, 0.3, 0.4]]).astype(np.float32)
>>> label = np.array(1).astype(np.int32)
>>> metric = OcclusionSensitivity()
>>> metric.clear()
>>> metric.update(model, test_data, label)
```

(continues on next page)

(continued from previous page)

```
>>> score = metric.eval()
>>> print(score)
[0.29999995    0.6    1.    0.9]
```

clear()
Clears the internal evaluation result.

eval()
Computes the occlusion_sensitivity.

Returns

A numpy ndarray.

Raises

RuntimeError –If the update method is not called first, an error will be reported.

update(**inputs*)
Updates input, including *model*, *y_pred* and *label*.

Parameters

inputs –*y_pred* and *label* are a Tensor, list or numpy.ndarray. *y_pred*: a batch of images to test, which could be 2D or 3D. *label*: classification labels to check for changes. *label* is normally the true label, but doesn't have to be. *model* is the neural network.

Raises

- **ValueError** –If the number of inputs is not 3.
- **RuntimeError** –If y_pred.shape[0] is not 1.
- **RuntimeError** –If the number of labels is different from the number of batches.

7.3.16 mindspore.train.Perplexity

class mindspore.train.**Perplexity**(*ignore_label=None*)
Computes perplexity. Perplexity is a measurement about how well a probability distribution or a model predicts a sample. A low perplexity indicates the model can predict the sample well. The function is shown as follows:

$$PP(W) = P(w_1 w_2 ... w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}}$$

Where *w* represents words in corpus. The root sign is the reciprocal of the probability of a sentence, and the better the sentence (with a higher probability), the lower the perplexity.

Parameters

ignore_label (*Union[int, None]*) –Index of an invalid label to be ignored when counting. If set to *None*, it will include all entries. Default: `None` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import Perplexity
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([1, 0, 1]))
>>> metric = Perplexity(ignore_label=None)
>>> metric.clear()
>>> metric.update(x, y)
>>> perplexity = metric.eval()
>>> print(perplexity)
2.231443166940565
```

clear()
Clears the internal evaluation result.

eval()
Returns the current evaluation result.

Returns

numpy.float64. The computed result.

Raises

RuntimeError –If the sample size is 0.

update(**inputs*)
Updates the internal evaluation result *preds* and *labels*.

Parameters

inputs –Input *preds* and *labels*. *preds* and *labels* are a *Tensor*, list or numpy.ndarray. *preds* is the predicted values, *labels* is the labels of the data. The shape of *preds* and *labels* are both (N, C).

Raises

- **ValueError** –If the number of the inputs is not 2.
- **RuntimeError** –If preds and labels have different lengths.
- **RuntimeError** –If label shape is not equal to pred shape.

7.3.17 mindspore.train.Precision

class mindspore.train.**Precision**(*eval_type='classification'*)
Calculates precision for classification and multilabel data.

The precision function creates two local variables, true_positive and false_positive, which are used to compute the precision. The calculation formula is:

$$\text{precision} = \frac{\text{true\_positive}}{\text{true\_positive} + \text{false\_positive}}$$

Parameters

eval_type (*str*) –'classification' or 'multilabel' are supported. See the update method below for what it does. Default: 'classification' .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import Precision
>>>
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([1, 0, 1]))
>>> metric = Precision('classification')
>>> metric.clear()
>>> metric.update(x, y)
>>> precision = metric.eval()
>>> print(precision)
[0.5 1. ]
```

clear()
Clears the internal evaluation result.

eval(*average=False*)
Computes the precision.

Parameters

average (*bool*) –Specify whether calculate the average precision. Default: `False` .

Returns

numpy.float64, the computed result.

update(**inputs*)
Updates the internal evaluation result with y_pred and y. In the multi-label cases, the elements of y and y_pred must be 0 or 1.

Parameters

inputs –Input y_pred and y. y_pred and y are Tensor, list or numpy.ndarray. For 'classification' evaluation type, y_pred is in most cases (not strictly) a list of floating numbers in range $[0, 1]$ and the shape is (N, C), where N is the number of cases and C is the number of categories. Shape of y can be (N, C) with values 0 and 1 if one-hot encoding is used or the shape is $(N,)$ with integer values if index of category is used. For 'multilabel' evaluation type, y_pred and y can only be one-hot encoding with values 0 or 1. Indices with 1 indicate positive category. The shape of y_pred and y are both (N, C).

Raises

ValueError –If the number of inputs is not 2.

7.3.18 mindspore.train.Recall

class mindspore.train.**Recall**(*eval_type='classification'*)
Calculates recall for classification and multilabel data.

The recall class creates two local variables, true_positive and false_negative, that are used to compute the recall. The calculation formula is:

$$\text{recall} = \frac{\text{true\_positive}}{\text{true\_positive} + \text{false\_negative}}$$

Note: In the multi-label cases, the elements of y and y_{pred} must be 0 or 1.

Parameters

eval_type (*str*) – `'classification'` or `'multilabel'` is supported. Default: `'classification'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import Recall
>>>
>>> x = Tensor(np.array([[0.2, 0.5], [0.3, 0.1], [0.9, 0.6]]))
>>> y = Tensor(np.array([1, 0, 1]))
>>> metric = Recall('classification')
>>> metric.clear()
>>> metric.update(x, y)
>>> recall = metric.eval()
>>> print(recall)
[1. 0.5]
```

clear()
Clears the internal evaluation result.

eval(*average=False*)
Computes the recall.

Parameters

average (*bool*) – Specify whether calculate the average recall. Default: `False`.

Returns

numpy.float64, the computed result.

update(**inputs*)
Updates the internal evaluation result with *y_pred* and *y*.

Parameters

inputs – Input *y_pred* and *y*. *y_pred* and *y* are a *Tensor*, a list or an array.

- For 'classification' scenario, *y_pred* is in most cases (not strictly) a list of floating numbers in range $[0, 1]$ and the shape is (N, C), where N is the number of cases and C is the number of categories. Shape of *y* can be (N, C) with values 0 and 1 if one-hot encoding is used or the shape is $(N,)$ with integer values if index of category is used.
- For 'multilabel' scenario, *y_pred* and *y* can only be one-hot encoding with values 0 or 1. Indices with 1 indicate positive category. The shape of *y_pred* and *y* are both (N, C).

Raises

ValueError – If the number of inputs is not 2.

7.3.19 mindspore.train.ROC

class mindspore.train.**ROC**(*class_num=None*, *pos_label=None*)

Calculates the ROC curve. It is suitable for solving binary classification and multi classification problems. In the case of multiclass, the values will be calculated based on a one-vs-the-rest approach.

Parameters

- **class_num** (*int*) –The number of classes. It is not necessary to provide this argument under the binary classification scenario. Default: None .
- **pos_label** (*int*) –Determine the integer of positive class. For binary problems, it is translated to 1 by default. For multiclass problems, this argument should not be set, as it will iteratively changed in the range [0, num_classes-1]. Default: None .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import ROC
>>>
>>> # 1) binary classification example
>>> x = Tensor(np.array([0.28, 0.55, 0.15, 0.05]))
>>> y = Tensor(np.array([0, 1, 2, 3]))
>>> metric = ROC(pos_label=2)
>>> metric.clear()
>>> metric.update(x, y)
>>> fpr, tpr, thresholds = metric.eval()
>>> print(fpr)
[0.         0.33333333 0.66666667 0.66666667 1.        ]
>>> print(tpr)
[0. 0. 0. 1. 1.]
>>> print(thresholds)
[1.55 0.55 0.28 0.15 0.05]
>>>
>>> # 2) multiclass classification example
>>> x = Tensor(np.array([[0.28, 0.55, 0.15, 0.05], [0.10, 0.20, 0.05, 0.05], [0.20, 0.05, 0.
↪15, 0.05],
...                      [0.05, 0.05, 0.05, 0.75]]))
>>> y = Tensor(np.array([0, 1, 2, 3]))
>>> metric = ROC(class_num=4)
>>> metric.clear()
>>> metric.update(x, y)
>>> fpr, tpr, thresholds = metric.eval()
>>> print(fpr)
[array([0., 0., 0.33333333, 0.66666667, 1.]), array([0., 0.33333333, 0.33333333, 1.]),
array([0., 0.33333333, 1.]), array([0., 0., 1.])]
>>> print(tpr)
[array([0., 1., 1., 1., 1.]), array([0., 0., 1., 1.]), array([0., 1., 1.]), array([0., 1., 1.
↪])]
>>> print(thresholds)
```

(continues on next page)

(continued from previous page)

```
[array([1.28, 0.28, 0.2, 0.1, 0.05]), array([1.55, 0.55, 0.2, 0.05]), array([1.15, 0.15, 0.
↪05]),
array([1.75, 0.75, 0.05])]
```

clear()
Clear the internal evaluation result.

eval()
Computes the ROC curve.

Returns

A tuple, composed of *fpr*, *tpr*, and *thresholds*.

- **fpr** (np.array) - False positive rate. In binary classification case, a fpr numpy array under different thresholds will be returned, otherwise in multiclass case, a list of fpr numpy arrays will be returned and each element represents one category.
- **tpr** (np.array) - True positive rates. n binary classification case, a tpr numpy array under different thresholds will be returned, otherwise in multiclass case, a list of tpr numpy arrays will be returned and each element represents one category.
- **thresholds** (np.array) - Thresholds used for computing fpr and tpr.

Raises

RuntimeError – If the update method is not called first, an error will be reported.

update(**inputs*)
Update state with predictions and targets.

Parameters

inputs – Input *y_pred* and *y*. *y_pred* and *y* are *Tensor*, list or numpy.ndarray. In most cases (not strictly), y_pred is a list of floating numbers in range $[0, 1]$ and the shape is (N, C), where N is the number of cases and C is the number of categories. y contains values of integers. The shape is (N, C) if one-hot encoding is used. Shape can also be $(N,)$ if category index is used.

7.3.20 mindspore.train.RootMeanSquareDistance

class mindspore.train.**RootMeanSquareDistance**(*symmetric=False*, *distance_metric='euclidean'*)
Computes the Root Mean Square Surface Distance from *y_pred* to *y* under the default setting.

Given two sets A and B, S(A) denotes the set of surface voxels of A, the shortest distance of an arbitrary voxel v to S(A) is defined as:

$$\operatorname{dis}(v, S(A)) = \min_{s_A \in S(A)} \|v - s_A\|$$

The Root Mean Square Surface Distance from set(B) to set(A) is:

$$RmsSurDis(B \to A) = \sqrt{\frac{\sum_{s_B \in S(B)} \operatorname{dis}^2(s_B, S(A))}{|S(B)|}}$$

Where the ||*|| denotes a distance measure. |*| denotes the number of elements.

The Root Mean Square Surface Distance from set(B) to set(A) and from set(A) to set(B) is:

$$RmsSurDis(A \leftrightarrow B) = \sqrt{\frac{\sum_{s_A \in S(A)} \operatorname{dis}(s_A, S(B))^2 + \sum_{s_B \in S(B)} \operatorname{dis}(s_B, S(A))^2}{|S(A)| + |S(B)|}}$$

Parameters

- **distance_metric** (*str*) −Three measurement methods are supported: "euclidean" (Euclidean Distance) , "chessboard" (Chessboard Distance, Chebyshev Distance) or "taxicab" (Taxicab Distance, Manhattan Distance). Default: "euclidean" .
- **symmetric** (*bool*) −Whether to calculate the symmetric average root mean square distance between *y_pred* and *y*. If False, only calculates $RmsSurDis(y\_pred, y)$ surface distance, otherwise, the mean of distance from *y_pred* to *y* and from *y* to *y_pred*, i.e. $RmsSurDis(y\_pred \leftrightarrow y)$ will be returned. Default: False .

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import RootMeanSquareDistance
>>>
>>> x = Tensor(np.array([[3, 0, 1], [1, 3, 0], [1, 0, 2]]))
>>> y = Tensor(np.array([[0, 2, 1], [1, 2, 1], [0, 0, 1]]))
>>> metric = RootMeanSquareDistance(symmetric=False, distance_metric="euclidean")
>>> metric.clear()
>>> metric.update(x, y, 0)
>>> root_mean_square_distance = metric.eval()
>>> print(root_mean_square_distance)
1.0000000000000002
```

clear()
Clears the internal evaluation result.

eval()
Calculate Root Mean Square Distance.

Returns

numpy.float64, root mean square surface distance.

Raises

RuntimeError −If the update method is not called first, an error will be reported.

update(**inputs*)
Updates the internal evaluation result 'y_pred', 'y' and 'label_idx'.

Parameters

inputs −Input 'y_pred', 'y' and 'label_idx'. 'y_pred' and 'y' are *Tensor*, list or numpy.ndarray. 'y_pred' is the predicted binary image. 'y' is the actual binary image. 'label_idx', the data type of *label_idx* is int.

Raises

- **ValueError** −If the number of the inputs is not 3.
- **TypeError** −If the data type of label_idx is not int or float.
- **ValueError** −If the value of label_idx is not in y_pred or y.
- **ValueError** −If y_pred and y have different shapes.

7.3.21 mindspore.train.Top1CategoricalAccuracy

class mindspore.train.**Top1CategoricalAccuracy**

Calculates the top-1 categorical accuracy. This class is a specialized class for TopKCategoricalAccuracy. Refer to *mindspore.train.TopKCategoricalAccuracy* for more details.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import Top1CategoricalAccuracy
>>>
>>> x = Tensor(np.array([[0.2, 0.5, 0.3, 0.6, 0.2], [0.1, 0.35, 0.5, 0.2, 0.],
...          [0.9, 0.6, 0.2, 0.01, 0.3]]), mindspore.float32)
>>> y = Tensor(np.array([2, 0, 1]), mindspore.float32)
>>> topk = Top1CategoricalAccuracy()
>>> topk.clear()
>>> topk.update(x, y)
>>> output = topk.eval()
>>> print(output)
0.0
```

7.3.22 mindspore.train.Top5CategoricalAccuracy

class mindspore.train.**Top5CategoricalAccuracy**

Calculates the top-5 categorical accuracy. This class is a specialized class for TopKCategoricalAccuracy. Refer to *mindspore.train.TopKCategoricalAccuracy* for more details.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.train import Top5CategoricalAccuracy
>>>
>>> x = Tensor(np.array([[0.2, 0.5, 0.3, 0.6, 0.2], [0.1, 0.35, 0.5, 0.2, 0.],
...             [0.9, 0.6, 0.2, 0.01, 0.3]]), mindspore.float32)
>>> y = Tensor(np.array([2, 0, 1]), mindspore.float32)
>>> topk = Top5CategoricalAccuracy()
>>> topk.clear()
>>> topk.update(x, y)
>>> output = topk.eval()
>>> print(output)
1.0
```

7.3.23 mindspore.train.TopKCategoricalAccuracy

class mindspore.train.**TopKCategoricalAccuracy**(*k*)

Calculates the top-k categorical accuracy.

Parameters

k (*int*) –Specifies the top-k categorical accuracy to compute.

Raises

- **TypeError** –If *k* is not int.
- **ValueError** –If *k* is less than 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.train import TopKCategoricalAccuracy
>>>
>>> x = Tensor(np.array([[0.2, 0.5, 0.3, 0.6, 0.2], [0.1, 0.35, 0.5, 0.2, 0.],
...          [0.9, 0.6, 0.2, 0.01, 0.3]]), mindspore.float32)
>>> y = Tensor(np.array([2, 0, 1]), mindspore.float32)
>>> topk = TopKCategoricalAccuracy(3)
>>> topk.clear()
>>> topk.update(x, y)
>>> output = topk.eval()
>>> print(output)
0.6666666666666666
```

clear()

Clear the internal evaluation result.

eval()

Computes the top-k categorical accuracy.

Returns

numpy.float64, computed result.

update(**inputs*)

Updates the internal evaluation result *y_pred* and *y*.

Parameters

inputs –Input *y_pred* and *y*. *y_pred* and *y* are Tensor, list or numpy.ndarray. *y_pred* is in most cases (not strictly) a list of floating numbers in range $[0, 1]$ and the shape is (N, C), where N is the number of cases and C is the number of categories. *y* contains values of integers. The shape is (N, C) if one-hot encoding is used. Shape can also be $(N,)$ if category index is used.

Note: The method *update* must receive input of the form (y_pred, y). If some samples have the same accuracy, the first sample will be chosen.

7.4 Utils

API Name	Description	Supported Platforms
mindspore.train.auc	Computes the AUC(Area Under the Curve) using the trapezoidal rule.	Ascend GPU CPU
mindspore.train.get_metric_fn	Gets the metric method based on the input name.	Ascend GPU CPU
mindspore.train.names	Gets all names of the metric methods.	Ascend GPU CPU
mindspore.train.rearrange_inputs	This decorator is used to rearrange the inputs according to its *indexes* attribute of the class.	Ascend GPU CPU

7.4.1 mindspore.train.auc

mindspore.train.**auc**(*x*, *y*, *reorder=False*)

Computes the AUC(Area Under the Curve) using the trapezoidal rule. This is a general function, given points on a curve, for computing the area under the ROC-curve.

Parameters

- **x** (*Union[np.array, list]*) –From the ROC curve(fpr), np.array with false positive rates. If multiclass, this is a list of such np.array, one for each class. The shape (N).
- **y** (*Union[np.array, list]*) –From the ROC curve(tpr), np.array with true positive rates. If multiclass, this is a list of such np.array, one for each class. The shape (N).
- **reorder** (*bool*) –If False, x must rise or fall monotonously. If True, x will be sorted in ascending order. Default: False.

Returns

float, the area under the ROC-curve.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore.train import ROC, auc
>>>
>>> y_pred = np.array([[3, 0, 1], [1, 3, 0], [1, 0, 2]])
>>> y = np.array([[0, 2, 1], [1, 2, 1], [0, 0, 1]])
>>> metric = ROC(pos_label=2)
>>> metric.clear()
>>> metric.update(y_pred, y)
>>> fpr, tpr, thre = metric.eval()
>>> output = auc(fpr, tpr)
>>> print(output)
0.5357142857142857
```

7.4.2 mindspore.train.get_metric_fn

mindspore.train.**get_metric_fn**(*name*, **args*, ***kwargs*)

Gets the metric method based on the input name.

Parameters

- **name** (*str*) −The name of metric method. Names can be obtained by *mindspore.train.names()* . object for the currently supported metrics.
- **args** −Arguments for the metric function.
- **kwargs** −Keyword arguments for the metric function.

Returns

Metric object, class instance of the metric method.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore.train import get_metric_fn
>>> metric = get_metric_fn('precision', eval_type='classification')
```

7.4.3 mindspore.train.names

mindspore.train.**names**()

Gets all names of the metric methods.

Returns

List, the name list of metric methods.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> ms.train.names()
['F1', 'acc', 'accuracy', 'auc', 'bleu_score', 'confusion_matrix', 'confusion_matrix_metric',
'cosine_similarity', 'dice', 'hausdorff_distance', 'loss', 'mae', 'mean_surface_distance',
 ↪'mse',
'occlusion_sensitivity', 'perplexity', 'precision', 'recall', 'roc', 'root_mean_square_
 ↪distance',
'top_1_accuracy', 'top_5_accuracy', 'topk']
```

7.4.4 mindspore.train.rearrange_inputs

mindspore.train.**rearrange_inputs**(*func*)

This decorator is used to rearrange the inputs according to its *indexes* attribute of the class.

This decorator is currently applied on the *update* of *mindspore.train.Metric*.

Parameters

func (*Callable*) – A candidate function to be wrapped whose input will be rearranged.

Returns

Callable, used to exchange metadata between functions.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore.train import rearrange_inputs
>>> class RearrangeInputsExample:
...     def __init__(self):
...         self._indexes = None
...
...     @property
...     def indexes(self):
...         return getattr(self, '_indexes', None)
...
...     def set_indexes(self, indexes):
...         self._indexes = indexes
...         return self
...
...     @rearrange_inputs
...     def update(self, *inputs):
...         return inputs
>>>
>>> rearrange_inputs_example = RearrangeInputsExample().set_indexes([1, 0])
>>> outs = rearrange_inputs_example.update(5, 9)
>>> print(outs)
(9, 5)
```

CHAPTER

EIGHT

MINDSPORE.PARALLEL

mindspore.parallel provides a large number of interfaces for automatic parallelization, including parallel base configuration, model loading and transformation, and functional parallel slicing.

The module import method is as follows:

```
from mindspore import parallel
```

8.1 Parallel Base Configuration

mindspore.parallel.auto_parallel.AutoParallel	Encapsulation of top-level Cells or functions to realize static graph parallelism for a single network.
mindspore.parallel.nn.GradAccumulation	Implementation of parallel gradient accumulation for static graphs.
mindspore.parallel.nn.MicroBatchInterleaved	Implement the static graph parallel multi-copy splitting function to enable concurrent computation and communication.
mindspore.parallel.nn.Pipeline	Specify the number of micro_batch for pipeline parallelism and the division rules for stage.
mindspore.parallel.nn.PipelineGradReducer	Functional training scenarios for gradient statute and accumulation of pipeline parallel.

8.1.1 mindspore.parallel.auto_parallel.AutoParallel

class mindspore.parallel.auto_parallel.**AutoParallel**(*network*, *parallel_mode='semi_auto'*)

Encapsulation of top-level Cells or functions to realize static graph parallelism for a single network.

Note:

- When using the *Model* API, the network passed to the *Model* must be wrapped with *AutoParallel*.
- When using *functional* API, the outermost layer must be wrapped with *AutoParallel*.
- When using *functional* API, data sinking mode are not currently supported.

Parameters

- **network** (*Union[Cell, Function]*) –Top-level cell or function in the forward network. Defines the core computational graph structure that will be parallelized.

- **parallel_mode** (*str, optional*) –Specifies the parallelization strategy engine. Available modes: "semi_auto", "sharding_propagation", "recursive_programming". Default: "semi_auto".
  - semi_auto: Achieves data and model parallelism by setting parallel strategies.
  - sharding_propagation: Automatic strategy propagation mode. Infers sharding strategies for non-annotated operators based on configured operator strategies. Dynamic shapes are not supported currently.
  - recursive_programming: Full automatic parallelization mode. Dynamically generates parallel strategies through recursive program analysis.

Supported Platforms:

Ascend

Examples

Note: You need to use the msrun command to run the following examples.

```
>>> import os
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import nn, ops
>>> from mindspore.communication import init, get_rank
>>> from mindspore.common.initializer import initializer
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.train import Model
>>> from mindspore.train import LossMonitor
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> ms.set_seed(1)
>>>
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>>
>>> def create_dataset(batch_size):
...     dataset_path = os.getenv("DATA_PATH")
...     dataset = ds.MnistDataset(dataset_path)
...     image_transforms = [
...         ds.vision.Rescale(1.0 / 255.0, 0),
...         ds.vision.Normalize(mean=(0.1307,), std=(0.3081,)),
...         ds.vision.HWC2CHW()
...     ]
...     label_transform = ds.transforms.TypeCast(ms.int32)
...     dataset = dataset.map(image_transforms, 'image')
...     dataset = dataset.map(label_transform, 'label')
...     dataset = dataset.batch(batch_size)
...     return dataset
>>>
>>> dataset = create_dataset(32)
>>>
>>> from mindspore import nn, ops, Parameter
>>> from mindspore.common.initializer import initializer, HeUniform
>>> import math
>>>
>>> class MatMulCell(nn.Cell):
```

(continues on next page)

(continued from previous page)

```
...     def __init__(self, param=None, shape=None):
...         super().__init__()
...         if shape is None:
...             shape = [28 * 28, 512]
...         weight_init = HeUniform(math.sqrt(5))
...         self.param = Parameter(initializer(weight_init, shape), name="param")
...         if param is not None:
...             self.param = param
...         self.print = ops.Print()
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         out = self.matmul(x, self.param)
...         self.print("out is:", out)
...         return out
>>>
>>> class Network(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.flatten = nn.Flatten()
...         self.layer1 = MatMulCell()
...         self.relu1 = nn.ReLU()
...         self.layer2 = nn.Dense(512, 512)
...         self.relu2 = nn.ReLU()
...         self.layer3 = nn.Dense(512, 10)
...
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.layer1(x)
...         x = self.relu1(x)
...         x = self.layer2(x)
...         x = self.relu2(x)
...         logits = self.layer3(x)
...         return logits
>>>
>>> import mindspore as ms
>>> from mindspore import nn, ops
>>> from mindspore.parallel.nn import Pipeline, PipelineGradReducer
>>> from mindspore.nn.utils import no_init_parameters
>>>
>>> with no_init_parameters():
>>>     net = Network()
>>>     optimizer = nn.SGD(net.trainable_params(), 1e-2)
>>>     pp_grad_reducer = PipelineGradReducer(optimizer.parameters, opt_shard=False)
>>>
>>> loss_fn = nn.CrossEntropyLoss()
>>> net_with_loss = Pipeline(nn.WithLossCell(net, loss_fn), 4, stage_config={"_backbone.
↪flatten":0,
>>>     "_backbone.layer1": 0, "_backbone.relu1": 0, "_backbone.layer2": 1,
>>>     "_backbone.relu2": 1, "_backbone.layer3": 1})
>>> parallel_net = AutoParallel(net_with_loss, parallel_mode="semi_auto")
>>> parallel_net.hsdp()
>>> parallel_net.pipeline(stages=2)
>>> parallel_net.dataset_strategy("data_parallel")
>>> parallel_net.save_param_strategy_file(f"/tmp/param_{get_rank()}.ckpt")
>>> parallel_net.set_group_ckpt_save_file(f"/tmp/comm_group_{get_rank()}.ckpt")
>>> parallel_net.dump_local_norm(f"/tmp/local_norm_{get_rank()}")
```

(continues on next page)

(continued from previous page)

```
>>> parallel_net.disable_strategy_file_only_for_trainable_params()
>>> parallel_net.enable_fp32_communication()
>>> parallel_net.enable_device_local_norm()
>>> parallel_net.enable_gradients_mean()
>>> parallel_net.disable_gradient_fp32_sync()
>>> parallel_net.disable_loss_repeated_mean()
>>>
>>> loss_monitor = LossMonitor(per_print_times=1)
>>> model = Model(network=parallel_net, optimizer=optimizer)
>>> model.train(epoch=2, train_dataset=dataset, callbacks=[loss_monitor])
```

comm_fusion(*config*)

Set fusion configuration of parallel communication operators.

Parameters

config (*dict*) – A dict contains the types and configurations for setting the communication fusion. Each communication fusion config has two keys: "mode" and "config". It supports following communication fusion types and configurations:

- openstate: Whether turn on the communication fusion or not. If *openstate* is *True*, turn on the communication fusion, otherwise, turn off the communication fusion. Default: *True*.
- allreduce: if communication fusion type is *allreduce*. The *mode* contains: *auto*, *size* and *index*. In *auto* mode, allreduce fusion is configured by gradients size, and the default fusion threshold is *64* MB. In 'size' mode, allreduce fusion is configured by gradients size manually, and the fusion threshold must be larger than *0* MB. In *index* mode, it is same as *all_reduce_fusion_config*.
- allgather: If communication fusion type is *allgather*. The *mode* contains: *auto*, *size*. In *auto* mode, AllGather fusion is configured by gradients size, and the default fusion threshold is *64* MB. In 'size' mode, AllGather fusion is configured by gradients size manually, and the fusion threshold must be larger than *0* MB.
- reducescatter: If communication fusion type is *reducescatter*. The *mode* contains: *auto* and *size*. Config is same as *allgather*.

Raises

TypeError – If the type of config is not dict.

Examples

```
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> parallel_net = AutoParallel(net, parallel_mode="semi_auto")
>>> comm_config = {"openstate": True, "allreduce": {"mode": "auto", "config": None}}
>>> net.comm_fusion(config=comm_config)
```

dataset_strategy(*config*)

Set dataset sharding strategy.

Parameters

config (*Union[str, tuple(tuple), tuple(Layout)]*) – The dataset sharding strategy. Default: "data_parallel". If you want to split dataset across devices, you can set the dataset strategy as "data_parallel". If you load whole batch datasets, you need to set the dataset strategy as "full_batch". For dataset load into net by dataset strategy like ds_stra((1, 8), (1, 8)),it requires using AutoParallel.dataset_strategy(ds_stra).Besides, dataset strategy also supports tuple of Layout.

Raises

- **TypeError** –When 'config' is not str type nor tuple type.
- **TypeError** –If 'config' is tuple type, but its element is not tuple type nor Layout type.
- **TypeError** –If 'config' is tuple type and its element is tuple type, the element in subtuple isn't int type.
- **ValueError** –If 'config' is None.
- **ValueError** –If the type of 'config' is str, but it's value is not 'full_batch' or 'data_parallel'.

disable_gradient_fp32_sync()
Disable convert tensor type from fp16 to fp32 before parameter gradients allreduce.

disable_loss_repeated_mean()
The mean operator is not executed backwards when the calculation is repeated.

disable_strategy_file_only_for_trainable_params()
By default, MindSpore only loads and saves trainable parameters. This API enables the loading and saving of non-trainable parameters as well.

dump_local_norm(*file_path*)
Enable local norm printing with disk storage only (no console output).

Parameters

file_path (*str*) –The path to save local_norm.

Raises

TypeError –If the type of 'file_path' is not str.

enable_device_local_norm()
Enable device local norm printing.

enable_fp32_communication()
Enable reduce operators (AllReduce, ReduceScatter) are forced to use the fp32 data type for communication during communication.

enable_gradients_mean()
Perform mean operator after allreduce of gradients in parallel mode.

hsdp(*shard_size=- 1*, *threshold=64*, *optimizer_level='level1'*)
Set optimizer parallel configs.

Parameters

- **shard_size** (*int, optional*) –Set the optimizer weight shard group size if you want to specific the maximum group size across devices when the parallel optimizer is enabled. The numerical range can be (0, device_num] or -1. Default value is -1, which means the optimizer weight shard group size will the data parallel group of each parameter.
- **threshold** (*int, optional*) –Set the threshold of parallel optimizer. When parallel optimizer is enabled, parameters with size smaller than this threshold will not be sharded across the devices. Parameter size = shape[0] * ···* shape[n] * size(dtype). Non-negative. Unit: KB. Default: 64.
- **optimizer_level** (*str, optional*) –optimizer_level configuration is used to specify the splitting level for optimizer sharding. It is important to note that the implementation of optimizer sharding in static graph is inconsistent with dynamic graph like megatron, but the memory optimization effect is the same. It must be one of [`level1`, `level2`, `level3`]. Default: `level1`.
  - level1: Splitting is performed on weights and optimizer state.
  - level2: Splitting is performed on weights, optimizer state, and gradients.

- level3: Splitting is performed on weights, optimizer state, gradients, additionally, before the backward pass, the weights are further applied with allgather communication to release the memory used by the forward pass allgather.

Raises

- **ValueError** –If the *shard_size* is not a positive integer or -1.
- **ValueError** –If *threshold* is not a positive integer or 0.
- **ValueError** –If *optimizer_level* is not one of the [`level1`, `level2`, `level3`].

load_operator_strategy_file(*file_path*)

Set the path to load strategy json when using sharding propagation.

> **Warning:** This is an experimental interface, may be changed or canceled in the future; This interface currently doesn't support loading strategies using layout.

Note:

- It only works when *parallel_mode=sharding_propagation*.
- When performing distributed training, users can first save the strategy using dryrun on a single device and then load strategy to perform distributed training.

Parameters

file_path (*str*) –Path to load parallel strategy json, must be an absolute path.

Raises

- **TypeError** –If the type of 'file_path' is not str.
- **KeyError** –When 'file_path' is not an absolute path.
- **KeyError** –When 'file_path' does not end in `".json"` .

Examples

```
>>> import math
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import nn, ops
>>> from mindspore.communication.management import init
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.common.initializer import initializer, HeUniform
>>>
>>> class ParallelNetwork(nn.Cell):
...     def __init__(self, strategy=None):
...         super().__init__()
...         self.flatten = ops.Flatten()
...         self.fc1_weight = ms.Parameter(initializer(HeUniform(math.sqrt(5)), shape=[
...             16, 10], dtype=ms.float32), name="fc1")
...         self.matmul1 = ops.MatMul().shard(strategy)
...         self.relu1 = ops.ReLU()
...
```

(continues on next page)

(continued from previous page)

```
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.matmul1(x, self.fc1_weight)
...         x = self.relu1(x)
...         return x
>>>
>>> init(backend_name='hccl')
>>> strategy = ((1, 1), (1, 2))
>>> net = ParallelNetwork(strategy)
>>> parallel_net = AutoParallel(net, parallel_mode='sharding_propagation')
>>> parallel_net.load_operator_strategy_file("/tmp/strategy.json")
```

load_param_strategy_file(*file_path*)

Set the path to load parallel sharding strategy file. By default, load strategy information for trainable parameters only.

Parameters

file_path (*str*) – The path to load parameter strategy checkpoint.

Raises

TypeError – If the type of 'file_path' is not str.

Examples

```
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> parallel_net = AutoParallel(net)
>>> parallel_net.load_param_strategy_file(file_path="./train_strategy.ckpt")
```

no_init_parameters_in_compile()

When enabled, the model weight parameters will not be initialized during the compilation process.

> **Warning:** This is an experimental interface, may be changed or canceled in the future.

Examples

```
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> parallel_net = AutoParallel(net, parallel_mode="semi_auto")
>>> parallel_net.no_init_parameters_in_compile()
```

pipeline(*stages=1*, *output_broadcast=False*, *interleave=False*, *scheduler='1f1b'*)

Configure the number of pipelin_dages, whether to broadcast the results, whether to enable interleaving scheduling, configure type of scheduler when using pipeline parallel.

Parameters

- **stages** (*int, optional*) – Set the stage information for pipeline parallelism This indicates how the devices are individually distributed on the pipeline. All devices will be divided into stages of pipine_dags. Default value: 1.
- **output_broadcast** (*bool, optional*) – When performing pipeline parallel inference, whether the result of the last stage is broadcasted to the other stages. Default value: False.

- **interleave** (*bool, optional*) –Whether to enable interleaving scheduling.
- **scheduler** (*str, optional*) –The type of scheduler

Raises

- **TypeError** –If the type of 'stages is not int.
- **ValueError** –When stages <= 0.
- **TypeError** –If the type of 'output_broadcast' is not bool.
- **TypeError** –If the type of 'interleave' is not bool.
- **TypeError** –If the type of 'scheduler' is not str.
- **ValueError** –If the type of 'scheduler' is not supported.

print_local_norm()

Print local norm value for auto parallel.

Examples

```
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> parallel_net = AutoParallel(net, parallel_mode="semi_auto")
>>> parallel_net.print_local_norm()
```

save_operator_strategy_file(*file_path*)

Set the path to save strategy json when using sharding propagation.

> **Warning:** This is an experimental interface, may be changed or canceled in the future; This interface currently doesn't support saving strategies using layout.

Note:

- It only works when *parallel_mode=sharding_propagation*.
- When performing distributed training, users can first save the strategy using dryrun on a single device and then load strategy to perform distributed training.

Parameters

file_path (*str*) –Path to save parallel strategy json, must be an absolute path.

Raises

- **TypeError** –If the type of 'file_path' is not str.
- **KeyError** –When 'file_path' is not an absolute path.
- **KeyError** –When 'file_path' does not end in `".json"` .

Examples

```
>>> import math
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import nn, ops
>>> from mindspore.communication.management import init
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.common.initializer import initializer, HeUniform
>>>
>>> class ParallelNetwork(nn.Cell):
...     def __init__(self, strategy=None):
...         super().__init__()
...         self.flatten = ops.Flatten()
...         self.fc1_weight = ms.Parameter(initializer(HeUniform(math.sqrt(5)), shape=[
...             16, 10], dtype=ms.float32), name="fc1")
...         self.matmul1 = ops.MatMul().shard(strategy)
...         self.relu1 = ops.ReLU()
...
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.matmul1(x, self.fc1_weight)
...         x = self.relu1(x)
...         return x
>>>
>>> init(backend_name='hccl')
>>> strategy = ((1, 1), (1, 2))
>>> net = ParallelNetwork(strategy)
>>> parallel_net = AutoParallel(net, parallel_mode='sharding_propagation')
>>> parallel_net.save_operator_strategy_file("/tmp/strategy.json")
```

save_param_strategy_file(*file_path*)

Set the path to save parallel sharding strategy file. By default, save strategy information for trainable parameters only.

Parameters

file_path (*str*) – The path where the parallel sharding strategy is saved.

Raises

TypeError – If the type of 'file_path' is not str.

set_group_ckpt_save_file(*file_path*)

Set the save path of the communication group.

Parameters

file_path (*str*) – The path to save parallel group checkpoint.

Raises

TypeError – If the type of 'file_path' is not str.

transformer_opt(*file_path*)

Check and set speedup config for auto parallel, configuration can refer to parallel_speed_up.json . If this parameter is set to None, it is disabled.

Parameters

file_path (*Union[str, None]*) – The path to the parallel speed up json file, configuration can refer to parallel_speed_up.json . If its value is None or '', it does not take effect. Default None.

- recomputation_communication_overlap (bool): Enable overlap between recompute ops and communication ops if True. Default: False.
- grad_matmul_communication_overlap (bool): Enable overlap between dw matmul and tensor parallel communication ops if True. Default: False.
- grad_fa_allgather_overlap (bool): Enable overlap between duplicated allgather by recomputing in sequence parallel and flashattentionscoregrad ops if True. Default: False.
- enable_communication_fusion (bool): Enable communication fusion to optimize the number of communication operator tasks if True. Default: False.
- grad_computation_allreduce_overlap (bool): Enable overlap between dx ops and data parallel communication ops if True. Currently, do not support O2 Default: False.
- computation_allgather_overlap (bool): Enable overlap between forward ops and optimizer parallel allgather communication if True. Currently, do not support O2 Default: False.
- computation_communication_fusion_level (int): Enable the fusion between compute and communicate. Default: `0`. Note: This function must be used with Ascend Training Solution 24.0.RC2 or later. This is an experimental configuration, may be changed or canceled in the future.
  - 0: Disable fusion.
  - 1: Apply fusion to forward nodes.
  - 2: Apply fusion to backward nodes.
  - 3: Apply fusion to all nodes.
- dataset_broadcast_opt_level (int): Optimize the scenario that the dataset repeated reading. Only support O0/O1 jit level. It doesn't work in O2 mode. Default: `0`.
  - 0: Disable this optimize.
  - 1: Optimize dataset reader between pipeline stage.
  - 2: Optimize dataset reader within pipeline stage.
  - 3: Optimize dataset reader with all scenes.
- allreduce_and_biasadd_swap (bool): Enable node execution order swap communication operators and add operators if `True`. Only 1-dimension bias node is supported. Default: `False`.
- enable_allreduce_slice_to_reducescatter (bool): Enable allreduce optimization. In the scenario where the batchmatmul model introduces allreduce in parallel, if the subsequent nodes are stridedslice operator with model parallel, allreduce will be optimized as reducescatter according to the identified patterns. Typical used in MoE module with groupwise alltoall. Default: `False`.
- enable_interleave_split_concat_branch (bool): Enable communication computation parallel optimization for branches formed by split and concat operators with `enable_interleave` attribute. It is typical used in MoE parallel scenario. After splitting the input data, each slice of data is processed by the MoE module, and then the branch results are concatenated. When the optimization is enable, communication and computation will be executed in parallel between branches. Default: `False`.
- enable_interleave_parallel_branch (bool): Enable communication computation parallel optimization for parallel branches with `parallel_branch` attribute in branches merge node. It is typical used in MoE parallel scenario with routed and shared expert. When the optimization is enable, communication and computation will be executed in parallel between branches. Default: `False`.

Examples

```
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>>
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = AutoParallel(net, parallel_mode="semi_auto")
>>> net.transformer_opt("./parallel_speed_up.json")
```

8.1.2 mindspore.parallel.nn.GradAccumulation

class mindspore.parallel.nn.**GradAccumulation**(*network*, *micro_size*)

Implementation of parallel gradient accumulation for static graphs.

Parameters

- **network** (Cell) –The target network to wrap.
- **micro_size** (*int*) –MicroBatch size.

Raises

- **TypeError** –The type of *network* is not cell.
- **TypeError** –If the type of *micro_size* is not int.
- **ValueError** –When *micro_size* is 0 or negative value.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore.parallel.nn import GradAccumulation
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = GradAccumulation(net, 4)
```

8.1.3 mindspore.parallel.nn.MicroBatchInterleaved

class mindspore.parallel.nn.**MicroBatchInterleaved**(*network*, *interleave_num=2*)

Implement the static graph parallel multi-copy splitting function to enable concurrent computation and communication. Application scenario: When there is model parallelism in semi-automatic mode and network, if the first slice data is calculating forward, the second slice data will execute the communication operators at the same time, to achieve the performance acceleration of communication and computing concurrency.

Parameters

- **network** (Cell) –The target network to wrap.
- **interleave_num** (*int, optional*) –split num of batch size. Default: 2 .

Inputs:

tuple[Tensor]. It's the same with the input of the *network* .

Outputs:

The wrapped input. The output of the input *network* should be a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = nn.MicroBatchInterleaved(net, 2)
```

8.1.4 mindspore.parallel.nn.Pipeline

class mindspore.parallel.nn.**Pipeline**(*network*, *micro_size*, *stage_config=None*)

Specify the number of micro_batch for pipeline parallelism and the division rules for stage.

Note: micro_size must be greater or equal to pipeline stages.

Parameters

- **network** (*Cell*) – The target network to wrap.
- **micro_size** (*int*) – MicroBatch size.
- **stage_config** (*dict, optional*) – Stage configuration for cell's execution in pipeline parallel. Default `None`.

Raises

- **TypeError** – The type of *net* is not cell.
- **TypeError** – If the type of *micro_size* is not int.
- **ValueError** – When *micro_size* is 0 or negative value.
- **KeyError** – *dict* cell name matching exception, there are remaining configuration items after traversing all *cell* under the current net.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel.nn import Pipeline
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> net = Pipeline(net, 4, stage_config={"cell_name_0": 0, "cell_name_1": 1})
```

8.1.5 mindspore.parallel.nn.PipelineGradReducer

class mindspore.parallel.nn.**PipelineGradReducer**(*parameters*, *scale_sense=1.0*, *opt_shard=None*)

Functional training scenarios for gradient statute and accumulation of pipeline parallel.

Parameters

- **parameters** (*list*) – the parameters to be updated.
- **scale_sense** (*float, optional*) – the scale sense of the gradient. Default: 1.0.
- **opt_shard** (*bool, optional*) – if use parallel optimizer, set opt_shard True. Default: None.

Raises

RuntimeError – If the mode is not graph mode.

Supported Platforms:

Ascend GPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Please see the rank table Startup for more details.

This example should be run with multiple devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import nn, ops, Tensor
>>> from mindspore.communication import init
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.reset_auto_parallel_context()
>>> init()
>>> ms.set_seed(1)
>>>
>>> class Network(nn.Cell):
...     def __init__(self, in_features, out_features, sens=1.0):
...         super().__init__()
...         self.layer1 = nn.Dense(in_features, 16)
...         self.relu1 = nn.ReLU()
...         self.layer2 = nn.Dense(16, 16)
...         self.relu2 = nn.ReLU()
```

(continues on next page)

(continued from previous page)

```
...         self.layer3 = nn.Dense(16, out_features)
...
...     def construct(self, x):
...         x = self.layer1(x)
...         x = self.relu1(x)
...         x = self.layer2(x)
...         x = self.relu2(x)
...         logits = self.layer3(x)
...         return logits
>>>
>>> size, in_features, out_features = 16, 32, 10
>>> net = Network(in_features, out_features)
>>> net.layer1.pipeline_stage = 0
>>> net.relu1.pipeline_stage = 0
>>> net.layer2.pipeline_stage = 0
>>> net.relu2.pipeline_stage = 1
>>> net.layer3.pipeline_stage = 1
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = nn.SGD(net.trainable_params(), 1e-2)
>>> net_with_loss = nn.Pipeline(nn.WithLossCell(net, loss_fn), 2)
>>> net_with_loss.set_train()
>>> def forward_fn(inputs, target):
...     loss = net_with_loss(inputs, target)
...     return loss
>>>
>>> grad_fn = ops.value_and_grad(forward_fn, None, net_with_loss.trainable_params())
>>> pp_grad_reducer = nn.PipelineGradReducer(optimizer.parameters)
>>>
>>> @ms.jit
>>> def train_one_step(inputs, target):
...     loss, grads = grad_fn(inputs, target)
...     grads = pp_grad_reducer(grads)
...     optimizer(grads)
...     return loss, grads
>>>
>>> parallel_net = AutoParallel(train_one_step, parallel_mode="semi_auto")
>>> parallel_net.pipeline(stages=2)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.ones([size, out_features]).astype(np.float32))
>>> loss, _ = train_one_step(inputs, label)
>>> print(loss)
46.36721
```

8.2 Model Loading and Transformation

mindspore.parallel.convert_checkpoints	Convert distributed checkpoint from source sharding strategy to destination sharding strategy for a rank.
mindspore.parallel.convert_checkpoint_by_rank	Convert distributed checkpoint from source sharding strategy to destination sharding strategy by rank for a network.
mindspore.parallel.load_distributed_checkpoint	Load checkpoint into net for distributed predication.

continues on next page

Table 2 – continued from previous page

mindspore.parallel.load_segmented_checkpoints	Load checkpoint info from a specified file.
mindspore.parallel.rank_list_for_convert	List of original distributed checkpoint rank index for obtaining the target checkpoint of a rank_id during the distributed checkpoint conversion.
mindspore.parallel.unified_safetensors	Merge multiple safetensor files into a unified safetensor file.

8.2.1 mindspore.parallel.convert_checkpoints

mindspore.parallel.**convert_checkpoints**(*src_checkpoints_dir*, *dst_checkpoints_dir*, *ckpt_prefix*, *src_strategy_file=None*, *dst_strategy_file=None*, *process_num=1*, *output_format='ckpt'*)

Convert distributed checkpoint from source sharding strategy to destination sharding strategy for a rank.

Note: The *src_checkpoints_dir* directory structure should be organized like "src_checkpoints_dir/rank_0/a.ckpt", the rank number should be set to a subdirectory and the checkpoint file is stored in this subdirectory. If multiple files exist in a rank directory, the last file in the lexicgraphic order would be selected.

The number of multiprocess settings is related to the size of the host, and it is not recommended to set it too large, otherwise it may cause freezing.

Parameters

- **src_checkpoints_dir** (*str*) –The source checkpoints directory.
- **dst_checkpoints_dir** (*str*) –The destination checkpoints directory to save the converted checkpoints.
- **ckpt_prefix** (*str*) –The destination checkpoint name prefix.
- **src_strategy_file** (*str, optional*) –Name of source sharding strategy file which saved by 'mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file'. when the 'src_strategy_file' is None, it means that the source sharding strategy is without any sharing for each parameter. Default:None.
- **dst_strategy_file** (*str, optional*) –Name of destination sharding strategy file which saved by 'mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file'. when the 'dst_strategy_file' is None, it means that the destination sharding strategy is without any sharing for each parameter. Default:None.
- **process_num** (*int, optional*) –Number of processes to use for parallel processing. Defaults: 1.
- **output_format** (*str, optional*) –Control the format of the output checkpoint after conversion. It can be set to either "ckpt" or "safetensors". Default: "ckpt".

Raises

- **ValueError** –*src_strategy_file* or *dst_strategy_file* is incorrect.
- **NotADirectoryError** –*src_checkpoints_dir* or *dst_checkpoints_dir* is not a directory.
- **ValueError** –The checkpoint file is missing in *src_checkpoints_dir*.
- **TypeError** –*src_strategy_file* or *dst_strategy_file* is not a string.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel import convert_checkpoints
>>> convert_checkpoints(src_checkpoints_dir, dst_checkpoints_dir, "dst_checkpoint",
...                     "./src_strategy.ckpt", "./dst_strategy.ckpt")
```

8.2.2 mindspore.parallel.convert_checkpoint_by_rank

mindspore.parallel.**convert_checkpoint_by_rank**(*rank_id*, *checkpoint_files_map*, *save_checkpoint_file_name*, *src_strategy_file=None*, *dst_strategy_file=None*)

Convert distributed checkpoint from source sharding strategy to destination sharding strategy by rank for a network.

Parameters

- **rank_id** (*int*) –The rank of which distributed checkpoint needs to be obtained after conversion.
- **checkpoint_files_map** (*dict*) –The checkpoint files map whose key is the rank id and the value is the checkpoint file name.
- **save_checkpoint_file_name** (*str*) –The file name to save the converted checkpoint.
- **src_strategy_file** (*str*) –Name of source sharding strategy file which saved by *mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file()*. when the *src_strategy_file* is None, it means that the source sharding strategy is without any sharing for each parameter. Default: `None`.
- **dst_strategy_file** (*str*) –Name of destination sharding strategy file which saved by *mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file()*. when the *dst_strategy_file* is `None`, it means that the destination sharding strategy is without any sharing for each parameter. Default: `None`.

Raises

- **ValueError** –*src_strategy_file* or *dst_strategy_file* is incorrect.
- **ValueError** –item in *checkpoint_files_map* is incorrect.
- **ValueError** –*save_checkpoint_file_name* is not end with ".ckpt".
- **TypeError** –*checkpoint_files_map* is not a dict.
- **TypeError** –*src_strategy_file* or *dst_strategy_file* is not a string.
- **TypeError** –*rank_id* is not an int.
- **TypeError** –*save_checkpoint_file_name* is not a string.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel import rank_list_for_convert, convert_checkpoint_by_rank
>>> dst_device_num = 8
>>> for rank_id in range(dst_device_num):
...     rank_list = rank_list_for_convert(rank_id, "./src_strategy.ckpt", "./dst_strategy.
↪ckpt")
...     checkpoint_files_map = {}
...     for rank in rank_list:
...         checkpoint_files_map[rank] = "./origin_checkpoint_rank{}/pangu{}-100_2.ckpt".
↪format(rank)
...     save_checkpoint_file_name = "./new_checkpoint_rank{}/pangu{}-100_2.ckpt".format(rank_
↪id)
...     convert_checkpoint_by_rank(rank_id, checkpoint_files_map, save_checkpoint_file_name,
...                                "./src_strategy.ckpt", "./dst_strategy.ckpt")
```

8.2.3 mindspore.parallel.load_distributed_checkpoint

mindspore.parallel.**load_distributed_checkpoint**(*network*, *checkpoint_filenames=None*, *predict_strategy=None*, *train_strategy_filename=None*, *strict_load=False*, *dec_key=None*, *dec_mode='AES-GCM'*, *format='ckpt'*, *unified_safetensors_dir=None*, *dst_safetensors_dir=None*, *rank_id=None*, *output_format='safetensors'*, *name_map=None*, *max_process_num=64*, *return_param_dict=False*)

Load checkpoint into net for distributed predication. Used in the case of distributed inference.

Note: *output_format* will only take effect when *format* is set to *safetensors* and *network* is set to *None*.

Parameters

- **network** (Cell) –Network for distributed predication, When the format is *safetensors*, the network parameter can be left blank or passed as None, and the interface will execute save mode.
- **checkpoint_filenames** (*list[str]*) –The name of Checkpoint files in order of rank id. Default: None .
- **predict_strategy** (*Union[dict, str]*) –Strategy of predication process. It means that using one device to predict when setting predict_strategy as None. Default: None .
- **train_strategy_filename** (*str*) –The filename of training strategy protocol buffer file. When train_strategy_filename is None, the training strategy file will be obtained from context.get_auto_parallel_context("strategy_ckpt_load_file"). Therefore, the training strategy file needs to be specified in at least one of them. Default: None .
- **strict_load** (*bool*) –Whether to strict load the parameter into net. If False , it will load parameter into net when parameter name's suffix in checkpoint file is the same as the parameter in the network. When the types are inconsistent, perform type conversion on the parameters of the same type, such as float32 to float16. Default: False .
- **dec_key** (*Union[None, bytes]*) –Byte type key used for decryption. If the value is None , the decryption is not required. Default: None .
- **dec_mode** (*str*) –Specifies the decryption mode, currently supports 'AES-GCM' , 'AES-CBC' and 'SM4-CBC' . This parameter is valid only when dec_key is not set to None . Default: 'AES-GCM' .
- **format** (*str*) –Input weight format to be loaded into the network. It can be set to either "ckpt" or "safetensors". Default: "ckpt".

- **unified_safetensors_dir** (*str*) –Directory of input weight files to be loaded into the network. Default: `None`.
- **dst_safetensors_dir** (*str*) –In the save mode scenario, the save directory for weights.
- **rank_id** (*int*) –The logical sequence number of the card. In non save mode, it is automatically obtained globally by initializing the network; In save mode, save the file according to the input sequence number. If it is not input, save the entire file.
- **output_format** (*str, optional*) –Control the format of the output checkpoint after conversion. It can be set to either "ckpt" or "safetensors". Default: "safetensors".
- **name_map** (*dict*) –The weight mapping dictionary will modify the weight names according to the mapping dictionary before loading or saving the segmented weights into the network. Default: None.
- **max_process_num** (*int*) –Maximum number of processes. Default: 64.
- **return_param_dict** (*bool*) –Whether to return the param_dict. Default: `False`.

Raises

- **TypeError** –The type of inputs do not match the requirements.
- **ValueError** –Failed to load checkpoint into net.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend devices, users need to prepare the rank table, set rank_id and device_id. Please see the rank table startup for more details.

For the CPU device, users need to write a dynamic cluster startup script, please see the Dynamic Cluster Startup .

```
>>> import os
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import nn, ops, train
>>> from mindspore.communication import init
>>> from mindspore.parallel import load_distributed_checkpoint
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.nn.utils import no_init_parameters
>>> from mindspore.common.initializer import initializer, One
>>> from mindspore.communication.management import get_group_size
>>>
>>> step_per_epoch = 4
>>> device_num = get_group_size()
>>>
>>> # Define the network structure.
>>> class Net(nn.Cell):
...     def __init__(self, matmul_size, strategy=None):
...         super().__init__()
...         self.matmul_weight = ms.Parameter(initializer(One(), matmul_size, ms.float32))
```

(continues on next page)

(continued from previous page)

```
...         self.matmul = ops.MatMul()
...         self.neg = ops.Neg()
...         if strategy is not None:
...             self.matmul.shard(strategy)
...
...     def construct(self, inputs):
...         x = self.matmul(inputs, self.matmul_weight)
...         x = self.neg(x)
...         return x
>>>
>>> # Create dataset.
>>> def get_dataset(*inputs):
...     def generate():
...         for _ in range(step_per_epoch):
...             yield inputs
...     return generate
>>>
>>> # Train network and save distributed checkpoint.
>>> def train_net():
...     ms.set_context(mode=ms.GRAPH_MODE)
...     init()
...     np.random.seed(1)
...     input_data = np.random.rand(16, 96).astype(np.float32)
...     label_data = np.random.rand(16, 16).astype(np.float32)
...     fake_dataset = get_dataset(input_data, label_data)
...     dataset = ds.GeneratorDataset(fake_dataset, ["input", "label"])
...
...     # Set parallel strategy.
...     strategy = ((1, 4), (4, 1))
...     with no_init_parameters():
...         network = Net(matmul_size=(96, 16), strategy=strategy)
...         net_opt = nn.Momentum(network.trainable_params(), 0.01, 0.9)
...
...     net_loss = nn.SoftmaxCrossEntropyWithLogits(reduction="mean")
...     network = AutoParallel(network, parallel_mode="semi_auto")
...     network.save_param_strategy_file(file_path="./train_strategy.ckpt")
...     model = ms.Model(network=network, loss_fn=net_loss, optimizer=net_opt)
...     ckpt_config = train.CheckpointConfig(keep_checkpoint_max=1, integrated_save=True)
...     global_rank_id = int(os.getenv("RANK_ID"))
...     ckpt_path = "./rank_{}_ckpt".format(global_rank_id)
...     ckpt_callback = train.ModelCheckpoint(prefix="parallel", directory=ckpt_path,␣
↪config=ckpt_config)
...     model.train(epoch=2, train_dataset=dataset, callbacks=[ckpt_callback], dataset_sink_
↪mode=False)
>>>
>>> # Load distributed checkpoint and test.
>>> def load_model():
...     ms.set_context(mode=ms.GRAPH_MODE)
...     init()
...     predict_data = ms.Tensor(np.random.randn(128, 96).astype(np.float32))
...     with no_init_parameters():
...         network = Net(matmul_size=(96, 16))
...         network = AutoParallel(network, parallel_mode="semi_auto")
...     network.dataset_strategy(config="full_batch")
...     train_strategy_file = "./train_strategy.ckpt"
...     network.save_param_strategy_file(file_path=train_strategy_file)
...     model = ms.Model(network)
```

(continues on next page)

(continued from previous page)

```
...     predict_layout = model.infer_predict_layout(ms.Tensor(predict_data))
...     ckpt_file_list = ["./rank_{}_ckpt/parallel-2_4.ckpt".format(i) for i in range(0,␣
↪device_num)]
...     load_distributed_checkpoint(network, ckpt_file_list, predict_layout, None)
...     predict_result = model.predict(predict_data)
...     print(predict_result)
>>>
>>> train_net()
>>> load_model()
[[-9.62929535e+00, -9.76258755e+00, -9.70192051e+00 ... -9.67151260e+00, -9.71998310e+00, -9.
↪64571190e+00],
[-4.63218540e-01, -4.07317460e-01, -3.78161550e-01 ... -3.95918339e-01, -2.87363172e-01, -3.
↪48693460e-01],
...
[-4.28075647e+00, -4.36630344e+00, -4.25664043e+00 ... -4.32012939e+00, -4.30337954e+00, -4.
↪27571440e+00]]
```

8.2.4 mindspore.parallel.load_segmented_checkpoints

mindspore.parallel.**load_segmented_checkpoints**(*ckpt_file_dir*, *net=None*, *strict_load=False*, *filter_prefix=None*, *dec_key=None*, *dec_mode='AES-GCM'*, *specify_prefix=None*, *choice_func=None*)

Load checkpoint info from a specified file. If the specified ckpt_file_dir path contains multiple checkpoint files, all checkpoint files will be loaded one by one and the combined dictionary will be return.

Note:

- *specify_prefix* and *filter_prefix* do not affect each other.
- If none of the parameters are loaded from checkpoint file, it will throw ValueError.
- *specify_prefix* and *filter_prefix* are in the process of being deprecated, *choice_func* is recommended instead. And using either of those two args will override *choice_func* at the same time.

Parameters

- **ckpt_file_dir** (*str*) – Checkpoint file directory.
- **net** (Cell) – The network where the parameters will be loaded. Default: None .
- **strict_load** (*bool*) – Whether to strict load the parameter into net. If False , it will load parameter into net when parameter name's suffix in checkpoint file is the same as the parameter in the network. When the types are inconsistent perform type conversion on the parameters of the same type, such as float32 to float16. Default: False .
- **filter_prefix** (*Union[str, list[str], tuple[str]]*) – Deprecated(see *choice_func*). Parameters starting with the filter_prefix will not be loaded. Default: None .
- **dec_key** (*Union[None, bytes]*) – Byte type key used for decryption. If the value is None , the decryption is not required. Default: None .
- **dec_mode** (*str*) – This parameter is valid only when dec_key is not set to None . Specifies the decryption mode, currently supports "AES-GCM" and "AES-CBC" and "SM4-CBC" . Default: "AES-GCM" .
- **specify_prefix** (*Union[str, list[str], tuple[str]]*) – Deprecated(see *choice_func*). Parameters starting with the specify_prefix will be loaded. Default: None .

- **choice_func** (*Union[None, function]*) –Input value of the function is a Parameter name of type string, and the return value is a bool. If returns `True` , the Parameter that matches the custom condition will be loaded. If returns `False` , the Parameter that matches the custom condition will be removed. Default: `None` .

Returns

Dict, key is parameter name, value is a Parameter or string. When the *append_dict* parameter of *mindspore.save_checkpoint()* and the *append_info* parameter of *mindspore.train.CheckpointConfig* are used to save the checkpoint, *append_dict* and *append_info* are dict types, and their value are string, then the return value obtained by loading checkpoint is string, and in other cases the return value is Parameter.

Raises

- **TypeError** –Input ckpt_file_dir is not a string.
- **ValueError** –Checkpoint file directory doesn't exist. Or it's not a directory
- **ValueError** –Checkpoint file's format is incorrect.
- **ValueError** –Parameter's dict is None after load checkpoint file.
- **TypeError** –The type of *specify_prefix* or *filter_prefix* is incorrect.

Supported Platforms:

`Ascend`

8.2.5 mindspore.parallel.rank_list_for_convert

mindspore.parallel.**rank_list_for_convert**(*rank_id*, *src_strategy_file=None*, *dst_strategy_file=None*)

List of original distributed checkpoint rank index for obtaining the target checkpoint of a rank_id during the distributed checkpoint conversion.

Parameters

- **rank_id** (*int*) –The rank of which distributed checkpoint needs to be obtained after conversion.
- **src_strategy_file** (*str*) –Name of source sharding strategy file which saved by *mindspore.parallel.auto_parallel.AutoParallel(cell).save_param_strategy_file(file_path)*. when the *src_strategy_file* is `None`, it means that the source sharding strategy is without any sharing for each parameter. Default: `None`.
- **dst_strategy_file** (*str*) –Name of destination sharding strategy file which saved by *mindspore.parallel.auto_parallel.AutoParallel(cell).save_param_strategy_file(file_path)*. when the *dst_strategy_file* is `None`, it means that the destination sharding strategy is without any sharing for each parameter. Default: `None`.

Returns

List, the rank list required for converting the distributed checkpoint of rank_id.

Raises

- **ValueError** –*src_strategy_file* or *dst_strategy_file* is incorrect.
- **TypeError** –*src_strategy_file* or *dst_strategy_file* is not a string.
- **TypeError** –*rank_id* is not an int.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel import rank_list_for_convert
>>> rank_id = 0
>>> rank_list = rank_list_for_convert(rank_id, "./src_strategy.ckpt", "./dst_strategy.ckpt")
>>> checkpoint_files_map = {}
>>> for rank in rank_list:
...     checkpoint_files_map[rank] = "./pangu{}-100_2.ckpt".format(rank)
```

8.2.6 mindspore.parallel.unified_safetensors

mindspore.parallel.**unified_safetensors**(*src_dir*, *src_strategy_file*, *dst_dir*, *merge_with_redundancy=True*, *file_suffix=None*, *max_process_num=64*, *choice_func=None*, *split_dst_file=()*)

Merge multiple safetensor files into a unified safetensor file.

Parameters

- **src_dir** (*str*) –Source weight saving directory.
- **src_strategy_file** (*str*) –Source weight segmentation strategy file.
- **dst_dir** (*str*) –Target save directory.
- **merge_with_redundancy** (*bool, optional*) –Whether the merged source weight files are de-duplicated and saved safetensors files. Default: `True`, indicating that the merged source weight files are complete.
- **file_suffix** (*str, optional*) –Specify the filename suffix for merging safetensors files. Default: `None`, meaning all safetensors files in the source weight directory will be merged.
- **max_process_num** (*int, optional*) –Maximum number of processes. Default: `64`.
- **choice_func** (*callable, optional*) –A callable function used to filter parameters or modify parameter names. The return value of the function must be of type str (string) or bool (boolean). Default: `None`.
- **split_dst_file** (*tuple, optional*) –execution, represented as a tuple containing two elements. The first element indicates the number of the current subtask, and the second element indicates the total number of tasks. This parameter supports splitting and executing tasks multiple times on a single machine, and also supports executing different subtasks on multiple machines respectively. Default: `()`.

Raises

ValueError –If the safetensors file of rank is missing.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore as ms
>>> src_dir = "/usr/safetensors/llama31B/4p_safetensors/"
>>> src_strategy_file = "/usr/safetensors/llama31B/strategy_4p.ckpt"
>>> dst_dir = "/usr/safetensors/llama31B/merge_llama31B_4p/"
>>> ms.parallel.unified_safetensors(src_dir, src_strategy_file, dst_dir)
```

8.3 Functional Parallel Slicing

mindspore.parallel.function.reshard	Converting a tensor from one distributed arrangement to another distributed arrangement.
mindspore.parallel.Layout	Topological abstraction describing cluster devices for tensor slice placement on the cluster.
mindspore.parallel.shard	Specify the input and output slicing strategy for a Cell or function.

8.3.1 mindspore.parallel.function.reshard

mindspore.parallel.function.**reshard**(*tensor*, *layout*)

Converting a tensor from one distributed arrangement to another distributed arrangement. The given layout must be type mindspore.parallel.Layout, can check *mindspore.parallel.Layout* for reference.

Note:

- In the Graph mode, this function can set the sharding propagation strategy of a tensor. For those tensor do not manually be set, their strategies are decided by the sharding strategy propagation algorithm automatically.
- In PyNative mode, you can use this method to arrange tensors in a cell (that is, cells that use Cell.shard/F.shard in PyNative mode) that is executed in parallel in graph mode.

Parameters

- **tensor** (Tensor) −The tensor to be set the sharding strategy.
- **layout** (Layout) −The layout to shard the tensor precisely, including the device arrangement (device_matrix) and the alias for the device matrix (alias_name).

Returns

Tensor. The mathematically equivalent of the input tensor.

Raises

- **TypeError** −If the type of input param *tensor* is not mindspore.Tensor.
- **TypeError** −If the type of input param *layout* is not mindspore.parallel.Layout.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start-up for more details.

This example should be run with 8 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import ops, nn, Tensor, context, Layout
>>> from mindspore.parallel.function import reshard
>>> from mindspore.nn.utils import no_init_parameters
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.communication import init
>>> context.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> class Network(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.matmul = ops.MatMul()
...         self.relu = ops.ReLU()
...     def construct(self, x, layout):
...         x = self.relu(x)
...         x_reshard = reshard(x, layout)
...         y = Tensor(np.ones(shape=(128, 128)), dtype=ms.float32)
...         x = self.matmul(x_reshard, y)
...         return x
>>> layout = Layout((4, 2), ("dp", "mp"))
>>> input_layout = layout("dp", "mp")
>>> with no_init_parameters():
...     net = Network()
>>> parallel_net = AutoParallel(net, parallel_mode='sharding_propagation')
>>> tensor = Tensor(np.ones(shape=(128, 128)), dtype=ms.float32)
>>> out = parallel_net(tensor, input_layout)
```

8.3.2 mindspore.parallel.Layout

class mindspore.parallel.**Layout**(*device_matrix*, *alias_name*, *rank_list=None*)

Topological abstraction describing cluster devices for tensor slice placement on the cluster.

Note:

- It is valid only in semi auto parallel or auto parallel mode.
- The multiplication result of the *device_matrix* must be equal to the device count in a pipeline stage.
- When the layout function is invoked to constructs a sharding strategy, each alias name is only allowed to be used once to shard a tensor.

Parameters

- **device_matrix** (*tuple*) – Describe the shape of devices arrangement, its element type is int.
- **alias_name** (*tuple*) – The alias name for each axis of device_matrix, its length shoits element type is string. When using "interleaved_parallel" as an alias name, the tensor would be split into multiple copies on the corresponding partition dimension on a single card.
- **rank_list** (*list, optional*) – Data is allocated to the device according to rank_list. Default: None.

Raises

- **TypeError** – *device_matrix* is not a tuple type.
- **TypeError** – *alias_name* is not a tuple type.

- **TypeError** – 'rank_list' is not a list type.
- **ValueError** – *device_matrix* length is not equal to *alias_name* length.
- **TypeError** – The element of *device_matrix* is not int type.
- **TypeError** – The element of *alias_name* is not a str type.
- **TypeError** – The element of *rank_list* is not int type.
- **ValueError** – The element of *alias_name* is an empty str.
- **ValueError** – The element of *alias_name* is "None".
- **ValueError** – *alias_name* contains repeated element.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel import Layout
>>> layout = Layout((2, 2, 2), ("dp", "sp", "mp"))
>>> layout0 = layout("dp", "mp")
>>> print(layout0.to_dict())
{'device_matrix': (2, 2, 2), 'tensor_map': (2, 0), 'interleaved_parallel': False,
'alias_name': {'dp', 'sp', 'mp'}, 'rank_list': [0, 1, 2, 3, 4, 5, 6, 7]}
>>> layout = Layout((2, 2, 2), ("dp", "sp", "interleaved_parallel"))
>>> layout1 = layout(("dp", "interleaved_parallel"), "sp")
```

to_dict()
Transform layout to a dictionary.

8.3.3 mindspore.parallel.shard

mindspore.parallel.**shard**(*fn*, *in_strategy*, *out_strategy=None*, *parameter_plan=None*, *device='Ascend'*, *level=0*)
Specify the input and output slicing strategy for a Cell or function. In PyNative mode, use this method to specify a Cell for distributed execution in graph mode. In Graph mode, use this method to specify distribution strategy for a Cell, strategy for others will be set by sharding propagation. in_strategy and out_strategy define the input and output layout respectively. in_strategy/out_strategy should be a tuple, each element of which corresponds to the desired layout of this input/output, and None represents data_parallel, which can refer to the description of *mindspore.ops.Primitive.shard()*. The parallel strategies of remaining operators are derived from the strategy specified by the input and output.

Note:

- It is valid only in semi auto parallel or auto parallel mode. In other parallel modes, strategies set here will be ignored.
- If the input contain Parameter, its strategy should be set in *in_strategy*.
- This method currently does not support dynamic shapes.

Parameters

- **fn** (*Union[Cell, Function]*) – Function to be executed in parallel. Its arguments and return value must be Tensor. If *fn* is a Cell with parameters, *fn* needs to be an instantiated object, otherwise its arguments cannot be accessed.

- **in_strategy** (*tuple*) –Define the layout of inputs, each element of the tuple should be a tuple(int) or tuple(mindspore.parallel.Layout). Tuple defines the layout of the corresponding input.
- **out_strategy** (*Union[tuple, None], optional*) –Define the layout of outputs similar with *in_strategy*. Default: `None` .
- **parameter_plan** (*Union[dict, None], optional*) –Define the layout for the specified parameters. Each element in dict defines the layout of the parameter like "param_name: layout". The key is a parameter name of type 'str'. The value is a 1-D integer tuple or a 1-D mindspore.parallel.Layout tuple, indicating the corresponding layout. If the parameter name is incorrect or the corresponding parameter has been set, the parameter setting will be ignored. Supported only when *fn* is a Cell with parameters. Default: `None` .
- **device** (*str, optional*) –Select a certain *device* target. It is not in use right now. Support ["CPU", "GPU", "Ascend"]. Default: `"Ascend"` .
- **level** (*int, optional*) –Option for parallel strategy infer algorithm, namely the object function, maximize computation over communication ratio, maximize speed performance, minimize memory usage etc. It is not in use right now. Support [0, 1, 2]. Default: `0` .

Returns

Function, return the function that will be executed under auto parallel process.

Raises

- **AssertionError** –If parallel mode is not "auto_parallel" nor "semi_auto_parallel".
- **AssertionError** –If device_target it not "Ascend" or "GPU".
- **TypeError** –If *in_strategy* is not a tuple.
- **TypeError** –If *out_strategy* is not a tuple or None.
- **TypeError** –If any element in *in_strategy* is not a tuple(int) or tuple(mindspore.parallel.Layout).
- **TypeError** –If any element in *out_strategy* is not a tuple(int) or tuple(mindspore.parallel.Layout).
- **TypeError** –If *parameter_plan* is not a dict or None.
- **TypeError** –If any key in *parameter_plan* is not a str.
- **TypeError** –If any value in *parameter_plan* is not a tuple(int) or a tuple(mindspore.parallel.Layout).
- **TypeError** –If *device* is not a str.
- **TypeError** –If *level* is not an integer.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, nn, ops
>>> from mindspore.communication import init
>>> from mindspore.parallel import shard
>>> from mindspore.parallel import Layout
>>> from mindspore.nn.utils import no_init_parameters
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> ms.set_context(mode=ms.GRAPH_MODE)
```

(continues on next page)

(continued from previous page)

```
>>> init()
>>>
>>> # Case 1: cell uses functional
>>> class BasicBlock(nn.Cell):
>>>     def __init__(self):
>>>         super(BasicBlock, self).__init__()
>>>         self.dense1 = nn.Dense(64, 64)
>>>         self.gelu = nn.GELU()
>>>         def my_add(x, y):
>>>             x = ops.abs(x)
>>>             return x + y
>>>         # shard a function with tuple(int) strategies
>>>         self.shard_my_add = shard(my_add, in_strategy=((2, 2), (1, 4)), out_strategy=((4,
 ↪ 1),))
>>>
>>>     def construct(self, x, u):
>>>         x = self.gelu(x)
>>>         y = self.gelu(u)
>>>         y = x * y
>>>         x = self.dense1(x)
>>>         x = self.shard_my_add(x, y)
>>>         return x
>>>
>>> class NetForward(nn.Cell):
>>>     def __init__(self):
>>>         super(NetForward, self).__init__()
>>>         self.block1 = BasicBlock()
>>>         self.block2 = BasicBlock()
>>>         self.matmul = ops.MatMul()
>>>
>>>     def construct(self, x, y):
>>>         x = self.matmul(x, y)
>>>         x = self.block1(x, x)
>>>         x = self.block2(x, x)
>>>         return x
>>>
>>> class Net(nn.Cell):
>>>     def __init__(self):
>>>         super(Net, self).__init__()
>>>         # setting cell sharding strategy and parameter_plan by tuple(int)
>>>         self.layer_net1 = NetForward()
>>>         self.layer_net1_shard = shard(self.layer_net1, in_strategy=((4, 2), (2, 1)),
...                                       parameter_plan={"self.layer_net1.block1.weight":
 ↪ (4, 1)})
>>>
>>>         # setting cell sharding strategy and parameter_plan by tuple(ms.Layout)
>>>         self.layer_net2 = NetForward()
>>>         layout = Layout((4, 2, 1), ("dp", "mp", "sp"))
>>>         in_layout = (layout("dp", "mp"), layout("mp", "sp"))
>>>         param_layout = layout("dp", "sp")
>>>         self.layer_net2_shard = shard(self.layer_net2, in_strategy=in_layout,
...                                       parameter_plan={"self.layer_net2.block2.weight":
 ↪ param_layout})
>>>         self.flatten = nn.Flatten()
>>>         self.layer1 = nn.Dense(64, 64)
>>>         self.layer2 = nn.Dense(64, 32)
>>>         self.add = ops.Add()
```

(continues on next page)

(continued from previous page)

```
>>>         self.matmul = ops.MatMul()
>>>
>>>     def construct(self, x, y):
>>>         x = self.flatten(x)
>>>         y = self.flatten(y)
>>>         x = self.layer1(x)
>>>         x = self.layer_net1_shard(x, y)
>>>         x = self.layer_net2_shard(x, y)
>>>         x = self.layer2(x)
>>>         x = self.matmul(x, Tensor(np.ones(shape=(32, 32)), dtype=ms.float32))
>>>         return x
>>>
>>> with no_init_parameters():
>>>     net = Net()
>>> x = Tensor(np.ones(shape=(64, 1, 8, 8)), dtype=ms.float32)
>>> y = Tensor(np.ones(shape=(64, 1, 8, 8)), dtype=ms.float32)
>>> parallel_net = AutoParallel(net, parallel_mode='sharding_propagation')
>>> parallel_net(x, y)
>>>
>>> # Case 2: function uses functional sharding
>>> def test_shard(x, y):
...     return x + y
>>> x = Tensor(np.ones(shape=(32, 10)), dtype=ms.float32)
>>> y = Tensor(np.ones(shape=(32, 10)), dtype=ms.float32)
>>> output = shard(test_shard, in_strategy=((4, 2), (4, 2)))(x, y)
>>> print(output.shape)
(32, 10)
```

8.4 Others

mindspore.parallel.build_searched_strategy	Extract the sharding strategy for each parameter in the network from the strategy file for distributed inference scenarios.
mindspore.parallel.merge_pipeline_strategys	Aggregate the sharding strategy files of all pipeline parallel sub-graphs to the destination file.
mindspore.parallel.parameter_broadcast	Broadcast parameter to other rank in data parallel dimension.
mindspore.parallel.restore_group_info_list	Extract rank list information from communication domain files.
mindspore.parallel.sync_pipeline_shared_parameters	Synchronization of shared weights between stages for pipeline parallel inference scenarios.

8.4.1 mindspore.parallel.build_searched_strategy

mindspore.parallel.**build_searched_strategy**(*strategy_filename*)

Extract the sharding strategy for each parameter in the network from the strategy file for distributed inference scenarios.

Parameters

strategy_filename (*str*) – Name of strategy file.

Returns

Dict, whose key is parameter name and value is slice strategy of this parameter.

Raises

- **ValueError** –Strategy file is incorrect.
- **TypeError** –*strategy_filename* is not a string.

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.parallel import build_searched_strategy
>>> strategy = build_searched_strategy("./strategy_train.ckpt")
```

8.4.2 mindspore.parallel.merge_pipeline_strategys

mindspore.parallel.**merge_pipeline_strategys**(*src_strategy_dirs*, *dst_strategy_file*)

Aggregate the sharding strategy files of all pipeline parallel subgraphs to the destination file.

Note: Strategy file of each pipeline stage should be included in src_strategy_dirs.

Parameters

- **src_strategy_dirs** (*str*) –The directory of strategy files including all pipeline stage which is saved by *mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file()*.
- **dst_strategy_file** (*str*) –The file merged strategy to save.

Raises

NotADirectoryError –*src_strategy_dirs* is not a directory.

Examples

```
>>> import mindspore as ms
>>> # src_strategy_dir/stra0.ckpt, src_strategy_dir/stra1.ckpt ... src_strategy_dir/stra127.
↪ckpt
>>> ms.parallel.merge_pipeline_strategys("./src_strategy_dir", "./dst_strategy.ckpt")
```

8.4.3 mindspore.parallel.parameter_broadcast

mindspore.parallel.**parameter_broadcast**(*net*, *layout*, *cur_rank=0*, *initial_rank=0*)

Broadcast parameter to other rank in data parallel dimension.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **net** (Cell) –The network where the parameters will be broadcasted.

- **layout** (*Dict*) –Parameter layout dictionary. Come from *mindspore.nn.Cell.parameter_layout_dict()* or read from file(for example: "strategy.ckpt" saved by using the *strategy_ckpt_config* parameter of *mindspore.parallel.auto_parallel.AutoParallel.save_param_strategy_file()*). The key is param name, the value is the layout of this parameter.
- **cur_rank** (*int, optional*) –current rank id. Default: `0`.
- **initial_rank** (*int, optional*) –Start rank id for each pipeline. Default: `0`.

Raises

- **ValueError** –*cur_rank* is not rank id of current rank.
- **ValueError** –*initial_rank* is not the start rank id of current pipeline stage.
- **ValueError** –Parameter name in *layout* can not be found in *mindspore.nn.Cell.parameters_dict()*.

Supported Platforms:

`Ascend`

Examples

```
>>> import os
>>> import mindspore as ms
>>> import mindspore.dataset as ds
>>> from mindspore import nn, ops
>>> from mindspore.communication import init
>>> from mindspore.common.initializer import initializer
>>> from mindspore.train import Model
>>> from mindspore.train.serialization import load_checkpoint, load_param_into_net
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> from mindspore.parallel import parameter_broadcast
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.runtime.set_memory(max_size="28GB")
>>> init()
>>> ms.set_seed(1)
>>> class Network(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.flatten = ops.Flatten()
...         self.fc1_weight = ms.Parameter(initializer("normal", [28*28, 512], ms.float32))
...         self.fc2_weight = ms.Parameter(initializer("normal", [512, 512], ms.float32))
...         self.fc3_weight = ms.Parameter(initializer("normal", [512, 10], ms.float32))
...         self.matmul1 = ops.MatMul()
...         self.relu1 = ops.ReLU()
...         self.matmul2 = ops.MatMul()
...         self.relu2 = ops.ReLU()
...         self.matmul3 = ops.MatMul()
...     def construct(self, x):
...         x = self.flatten(x)
...         x = self.matmul1(x, self.fc1_weight)
...         x = self.relu1(x)
...         x = self.matmul2(x, self.fc2_weight)
...         x = self.relu2(x)
...         logits = self.matmul3(x, self.fc3_weight)
...         return logits
>>> net = Network()
```

(continues on next page)

(continued from previous page)

```
>>> net.matmul1.shard(((2, 4), (4, 1)))
>>> net.relu1.shard(((4, 1),))
>>> net.matmul2.shard(((1, 8), (8, 1)))
>>> net.relu2.shard(((8, 1),))
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> optim = nn.SGD(net.trainable_params(), 1e-2)
>>> loss = nn.CrossEntropyLoss()
>>> parallel_net = AutoParallel(net)
>>> model = Model(parallel_net, loss_fn=loss, optimizer=optim)
>>> model.train(1, dataset)
>>> ms.save_checkpoint(net, "./simple.ckpt", False)
>>> layout = model.train_network.parameter_layout_dict
>>> param_dict = load_checkpoint("./simple.ckpt")
>>> load_param_into_net(net, param_dict)
>>> rank_id = os.environ["RANK_ID"]
>>> parameter_broadcast(model.train_network, layout, int(rank_id), 0)
>>> class LossCallBack(Callback):
...     def step_end(self, run_context):
...         cb_params = run_context.original_args()
...         print("step end, cur step num: ", cb_params.cur_step_num, flush=True)
>>> model.train(1, dataset, callbacks=[LossCallBack()])
```

8.4.4 mindspore.parallel.restore_group_info_list

mindspore.parallel.**restore_group_info_list**(*group_info_file_name*)

Extract rank list information from communication domain files. To save the group info file, please export GROUP_INFO_FIL environment variables like "export GROUP_INFO_FILE=/data/group_info.pb".

Parameters

group_info_file_name (*str*) – Name of group information file.

Returns

List, the rank list.

Raises

- **ValueError** – group information file is incorrect.
- **TypeError** – *group_info_file_name* is not str.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore as ms
>>> from mindspore.parallel import restore_group_info_list
>>> ms.restore_list = restore_group_info_list("./group_info.pb")
```

8.4.5 mindspore.parallel.sync_pipeline_shared_parameters

mindspore.parallel.**sync_pipeline_shared_parameters**(*net*)

Synchronization of shared weights between stages for pipeline parallel inference scenarios. For example, *embedding table* is shared by *WordEmbedding* layer and *LMHead* layer, which are usually split into different stages. It is necessary to perform synchronization after *embedding table* changes.

Note: The network should be compiled before shared parameters are synchronized in the pipeline parallel stage.

Parameters

net (Cell) – the inference network.

Raises

TypeError – *net* is not in Cell type.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For the Ascend device, users need to write a dynamic cluster startup script, please see the Dynamic Cluster Startup .

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication.management as D
>>> from mindspore import lazy_inline, context, nn, ops, Parameter, Tensor
>>> from mindspore.parallel.auto_parallel import AutoParallel
>>> context.set_context(mode=context.GRAPH_MODE)
>>> class Embedding(nn.Cell):
...     def __init__(self, shape):
...         super().__init__()
...         self.w = Parameter(Tensor(np.ones(shape), ms.float32), name='w')
...         self.matmul = ops.MatMul().shard(((1, 1), (1, 1)))
...     def construct(self, x):
...         return self.matmul(x, self.w), self.w
...
>>> class LMHead(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.matmul = ops.MatMul(transpose_b=True).shard(((1, 1), (1, 1)))
```

(continues on next page)

(continued from previous page)

```
...     def construct(self, x, w):
...         return self.matmul(x, w)
...
>>> class Network(nn.Cell):
...     @lazy_inline
...     def __init__(self):
...         super().__init__()
...         shape = (4, 4)
...         self.word_embedding = Embedding(shape)
...         self.lm_head = LMHead()
...         self.word_embedding.pipeline_stage = 0
...         self.lm_head.pipeline_stage = 1
...     def construct(self, x):
...         x, embed = self.word_embedding(x)
...         return self.lm_head(x, embed)
...
>>> class PipelineCellInference(nn.Cell):
...     def __init__(self, network, micro_batch_num):
...         super().__init__()
...         self.network = network
...         self.micro_batch_num = micro_batch_num
...         self.concat = ops.Concat()
...     def construct(self, x):
...         ret = ()
...         for i in range(self.micro_batch_num):
...             micro_batch_size = x.shape[0] // self.micro_batch_num
...             start = micro_batch_size * i
...             end = micro_batch_size * (i + 1)
...             micro_input = x[start:end]
...             y = self.network(micro_input)
...             ret = ret + (y,)
...         ret = self.concat(ret)
...         return ret
>>> D.init()
>>> net = Network()
>>> net = PipelineCellInference(net, 2)
>>> net.set_train(False)
>>> x = Tensor(np.ones((2, 4)), ms.float32)
>>> net.compile(x)
>>> pp_net = AutoParallel(net, parallel_mode="semi_auto")
>>> pp_net.full_batch = True
>>> pp_net.pipeline(stages=2, scheduler="1f1b")
>>> ms.parallel.sync_pipeline_shared_parameters(pp_net)
>>> print(pp_net.network.network.word_embedding.w.asnumpy())
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

CHAPTER

NINE

MINDSPORE.RUNTIME

Runtime encapsulates interfaces for executor, memory, stream and event. MindSpore abstracts the corresponding modules from different backends, allowing users to schedule hardware resources at the Python layer.

9.1 Executor

mindspore.runtime.set_cpu_affinity	Enable thread-level core binding to assign specific CPU cores to MindSpore's main modules (main thread, pynative, runtime, minddata), to prevent unstable performance caused by MindSpore's threads seizing CPU.
mindspore.runtime.launch_blocking	Indicates that synchronizing the execution of the startup device reduces the execution performance of the program.
mindspore.runtime.dispatch_threads_num	Set the threads number of runtime used.
mindspore.runtime.set _kernel_launch_group	O0 mode supports operator batch parallel delivery interface, supports enabling parallel delivery, and configures parallel number.

9.1.1 mindspore.runtime.set_cpu_affinity

mindspore.runtime.**set_cpu_affinity**(*enable_affinity*, *affinity_cpu_list=None*)

Enable thread-level core binding to assign specific CPU cores to MindSpore's main modules (main thread, pynative, runtime, minddata), to prevent unstable performance caused by MindSpore's threads seizing CPU.

Note:

- Provides two binding modes: 1. Automatically generates binding policies based on available CPUs, NUMA nodes, and device resources in the environment to bind cores at thread level. 2. Thread-level bonding based on customized bonding policies passed in by *affinity_cpu_list*.
- The automated bind-core policy generation scenario invokes system commands to obtain CPU, NUMA node, and device resources on the environment, and some commands cannot be executed successfully due to environment differences; the automated bind-core policy generated will vary according to the resources available on the environment:
  1. *cat /sys/fs/cgroup/cpuset/cpuset.cpus*, to obtain the available CPU resources on the environment; if the execution of this command fails, the bind-core function will not take effect.
  2. *npu-smi info -m*, get the available NPU resources on the environment; if the execution of this command fails, the bind-core policy will be generated only based on the available CPU resources, without considering the device affinity.
  3. *npu-smi info -t board -i {NPU_ID} -c {CHIP_ID}*, get NPU details based on the logical ID of the device; if the execution of this command fails, the bind-core policy is generated based on the available CPU resources only, regardless of device

affinity.

4. *lspci -s {PCIe_No} -vvv*, get the hardware information of the device on the environment; if the execution of this command fails, the bind-core policy is generated only based on the available CPU resources, without considering the device affinity.
5. *lscpu*, get information about CPUs and NUMA nodes on the environment; if the execution of this command fails, only the available CPU resources are used to generate the bind-core policy, without considering the device affinity.

Parameters

- **enable_affinity** (*bool*) –Switches on/off thread-level core binding.
- **affinity_cpu_list** (*dict, optional*) –Specifies a customized bind-core policy. The key to be passed into the dict needs to be in string `"deviceX"` format, and the value needs to be in list `["cpuidX-cpuidY"]` format. Default: `None`, i.e., use the bind-core policy generated automatically based on the environment. It is allowed to pass the empty dict `{}`, in which case the bind-core policy generated automatically based on the environment will be used.

Raises

- **TypeError** –The parameter *enable_affinity* is not a boolean.
- **TypeError** –The parameter *affinity_cpu_list* is neither a dictionary nor a `None`.
- **ValueError** –The key of parameter *affinity_cpu_list* is not a string.
- **ValueError** –The key of parameter *affinity_cpu_list* is not in `"deviceX"` format.
- **ValueError** –The parameter *affinity_cpu_list* has a value that is not a list.
- **ValueError** –The element in value of parameter *affinity_cpu_list* is not a string.
- **ValueError** –The element in value for parameter *affinity_cpu_list* does not match `["cpuidX-cpuidY"]`.
- **RuntimeError** –Automatically generated binding policy or customized binding policy scenario where the number of CPU cores assigned to each device is less than 7.
- **RuntimeError** –A custom-specified binding policy scenario where the CPU assigned to a device is not available in the environment.
- **RuntimeError** –The *mindspore.runtime.set_cpu_affinity* API is called repeatedly.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.runtime.set_cpu_affinity(True)
>>>
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.runtime.set_cpu_affinity(True, {"device0":["0-9"],"device1":["10-15","20-29"],"device2
↪":["35-45"]})
```

9.1.2 mindspore.runtime.launch_blocking

mindspore.runtime.**launch_blocking**()

Indicates that synchronizing the execution of the startup device reduces the execution performance of the program.

- In the initial state when this interface is not called, the operator executes asynchronously on the device. In this case, when an error occurs in the execution of the operator, it will not be possible to locate the position of the particular error script code.
- When this interface is called, the operator is executed in a synchronized manner on the device. At this point, when an error occurs in the execution of the operator, the location of the erroneous script code can be located based on the error call stack.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.runtime.launch_blocking()
```

9.1.3 mindspore.runtime.dispatch_threads_num

mindspore.runtime.**dispatch_threads_num**(*threads_num*)

Set the threads number of runtime used.

The framework set the runtime number of threads are 5 by default.

Parameters

threads_num (*int*) – The threads number of runtime used.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.runtime.dispatch_threads_num(6)
```

9.1.4 mindspore.runtime.set_kernel_launch_group

mindspore.runtime.**set_kernel_launch_group**(*thread_num=2*, *kernel_group_num=8*)

O0 mode supports operator batch parallel delivery interface, supports enabling parallel delivery, and configures parallel number.

Parameters

- **thread_num** (*int, optional*) – The number of concurrent threads, generally not recommended to increase. The *thread_num* and the number of threads configured by the existing interface mindspore.runtime.dispatch_threads_num are independent of each other. Default value is 2.
- **kernel_group_num** (*int, optional*) – Total number of operator groups, kernel_group_num/thread_num groups per thread. Default value is 8.

Examples

```
>>> import mindspore as ms
>>> ms.runtime.set_kernel_launch_group(thread_num=2, kernel_group_num=8)
```

9.2 Memory

mindspore.runtime.max_memory_allocated	Returns the peak memory size of the memory pool actually occupied by Tensor since the process was started.
mindspore.runtime.max_memory_reserved	Returns the peak value of the total memory managed by the memory pool since the process was started.
mindspore.runtime.memory_allocated	Returns the actual memory size currently occupied by Tensor.
mindspore.runtime.memory_reserved	Returns the total amount of memory currently managed by the memory pool.
mindspore.runtime.memory_stats	Returns status information queried from the memory pool.
mindspore.runtime.memory_summary	Returns readable memory pool status information.
mindspore.runtime.reset _max_memory_reserved	Reset the peak memory size managed by the memory pool.
mindspore.runtime.reset _max_memory_allocated	Reset the peak memory size of the memory pool actually occupied by Tensor.
mindspore.runtime.reset _peak_memory_stats	Reset the "peak" stats tracked by memory manager.
mindspore.runtime.empty_cache	Empty cache in the memory pool.
mindspore.runtime.set_memory	Set the memory parameters of runtime device memory management that is implemented using a memory pool.

9.2.1 mindspore.runtime.max_memory_allocated

mindspore.runtime.**max_memory_allocated**()

Returns the peak memory size of the memory pool actually occupied by Tensor since the process was started.

Note:

- For the *CPU* backend, 0 is always returned.

Returns

int, in Byte.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.max_memory_allocated())
1536
```

9.2.2 mindspore.runtime.max_memory_reserved

mindspore.runtime.**max_memory_reserved**()

Returns the peak value of the total memory managed by the memory pool since the process was started.

Note:

- For the *CPU* backend, 0 is always returned.

Returns

int, in Byte.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.max_memory_reserved())
1073741824
```

9.2.3 mindspore.runtime.memory_allocated

mindspore.runtime.**memory_allocated**()

Returns the actual memory size currently occupied by Tensor.

Note:

- For the *CPU* backend, 0 is always returned.

Returns

int, in Byte.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.memory_allocated())
1024
```

9.2.4 mindspore.runtime.memory_reserved

`mindspore.runtime.`**`memory_reserved`**`()`

Returns the total amount of memory currently managed by the memory pool.

Note:

- For the *CPU* backend, 0 is always returned.

Returns

int, in Byte.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.memory_reserved())
1073741824
```

9.2.5 mindspore.runtime.memory_stats

mindspore.runtime.**memory_stats**()

Returns status information queried from the memory pool.

Note: For the *CPU* backend, a dictionary with empty data is always returned.

Returns

dict, the queried memory information.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.memory_stats())
{'total_reserved_memory': 1073741824, 'total_allocated_memory': 1024, 'total_idle_memory':␣
↪1073740800,
'total_eager_free_memory': 0, 'max_reserved_memory': 1073741824, 'max_allocated_memory':␣
↪1536,
'common_mem_pool_stats': {'block_unit_size': 1073741824, 'block_counts': 1, 'blocks_info':
{<capsule object NULL at 0x7f7e8c27b030>: {'block_stream_id': 0, 'block_memory_size':␣
↪1073741824}}},
'persistent_mem_pool_stats': {'block_unit_size': 1073741824, 'block_counts': 0, 'blocks_info
↪': {}}}
```

9.2.6 mindspore.runtime.memory_summary

mindspore.runtime.**memory_summary**()

Returns readable memory pool status information.

Returns

str, readable memory pool status information in tabular form.

Supported Platforms:

Ascend GPU CPU

9.2.7 mindspore.runtime.reset_max_memory_reserved

mindspore.runtime.**reset_max_memory_reserved**()

Reset the peak memory size managed by the memory pool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.max_memory_reserved())
1073741824
>>> ms.runtime.reset_max_memory_reserved()
>>> print(ms.runtime.max_memory_reserved())
0
```

9.2.8 mindspore.runtime.reset_max_memory_allocated

mindspore.runtime.**reset_max_memory_allocated**()

Reset the peak memory size of the memory pool actually occupied by Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.max_memory_allocated())
1536
>>> ms.runtime.reset_max_memory_allocated()
>>> print(ms.runtime.max_memory_allocated())
0
```

9.2.9 mindspore.runtime.reset_peak_memory_stats

mindspore.runtime.**reset_peak_memory_stats**()

Reset the "peak" stats tracked by memory manager.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.runtime.max_memory_reserved())
1073741824
>>> print(ms.runtime.max_memory_allocated())
1536
>>> ms.runtime.reset_peak_memory_stats()
>>> print(ms.runtime.max_memory_reserved())
0
>>> print(ms.runtime.max_memory_allocated())
0
```

9.2.10 mindspore.runtime.empty_cache

mindspore.runtime.**empty_cache**()

Empty cache in the memory pool.

Note:

- Empty cache help reduce the fragmentation of device memory.
- Support Atlas A2 series products.

Supported Platforms:

Ascend

9.2.11 mindspore.runtime.set_memory

mindspore.runtime.**set_memory**(*init_size='2GB'*, *increase_size='2GB'*, *max_size='1024GB'*, *optimize_level='O0'*, *huge_page_reserve_size='0GB'*)

Set the memory parameters of runtime device memory management that is implemented using a memory pool.

The framework will set all the args by default as follows.

Parameters

- **init_size** (*str*) –The init size of memory pool. The format is "xxGB". Default: 2GB .

- **increase_size** (*str*) –The increase size of memory pool. When the current memory pool has no enough memory, the memory pool will be expanded by this value. The format is "xxGB". Default: `2GB` .
- **max_size** (*str*) –The maximum memory available for memory pool. The actual used memory size is the minimum of the available memory of the device and max_device_memory. The format is "xxGB". Default is the maximum available memory of the device, expressed as `1024GB`.
- **optimize_level** (*str*) –The memory optimize level. The value must be in ['O0', 'O1']. Default: `O0` .
- **huge_page_reserve_size** (*str*) –The reserved size of huge page memory. The format is "xxGB". Default: `0GB`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.runtime.set_memory("10GB", "2GB", "60GB", "O1", "0GB")
```

9.3 Stream

mindspore.runtime.communication_stream	Return communication stream on this device.
mindspore.runtime.current_stream	Return current stream used on this device.
mindspore.runtime.default_stream	Return default stream on this device.
mindspore.runtime.set_cur_stream	Sets the current stream.This is a wrapper API to set the stream.
mindspore.runtime.synchronize	Synchronize all streams on current device.(Each MindSpore process only occupies one device)
mindspore.runtime.Stream	Wrapper around a device stream.
mindspore.runtime.StreamCtx	Context-manager that selects a given stream.

9.3.1 mindspore.runtime.communication_stream

mindspore.runtime.**communication_stream**()

Return communication stream on this device.

Returns

stream (Stream), communication stream.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 0)
>>> ms.runtime.communication_stream()
Stream(device_name=Ascend, device_id:0, stream id:1)
```

9.3.2 mindspore.runtime.current_stream

mindspore.runtime.**current_stream**()

Return current stream used on this device.

Returns

stream (Stream), current stream.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 0)
>>> cur_stream = ms.runtime.current_stream()
>>> assert cur_stream == ms.runtime.default_stream()
```

9.3.3 mindspore.runtime.default_stream

mindspore.runtime.**default_stream**()

Return default stream on this device.

Returns

stream (Stream), default stream.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 0)
>>> cur_stream = ms.runtime.current_stream()
>>> assert cur_stream == ms.runtime.default_stream()
```

9.3.4 mindspore.runtime.set_cur_stream

mindspore.runtime.**set_cur_stream**(*stream*)

Sets the current stream.This is a wrapper API to set the stream. Usage of this function is discouraged in favor of the `stream` context manager.

Parameters

stream (Stream) –selected stream. This function is a no-op if this argument is None.

Raises

TypeError –If 'stream' is neither a *mindspore.runtime.Stream* nor a None.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 0)
>>> cur_stream = ms.runtime.current_stream()
>>> assert cur_stream == ms.runtime.default_stream()
>>> s1 = ms.runtime.Stream()
>>> ms.runtime.set_cur_stream(s1)
>>> assert ms.runtime.current_stream() == s1
>>> ms.runtime.set_cur_stream(ms.runtime.default_stream())
```

9.3.5 mindspore.runtime.synchronize

mindspore.runtime.**synchronize**()

Synchronize all streams on current device.(Each MindSpore process only occupies one device)

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1024, 2048]), ms.float32)
>>> b = Tensor(np.ones([2048, 4096]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
>>> ms.runtime.synchronize()
>>> assert s1.query()
```

9.3.6 mindspore.runtime.Stream

class mindspore.runtime.**Stream**(*priority=0*, ***kwargs*)

Wrapper around a device stream.

A device stream is a linear sequence of execution that belongs to a specific device, independent from other streams.

Parameters

- **priority** (*int, optional*) – priority of the stream, lower numbers represent higher priorities. By default, streams have priority `0`.
- **kwargs** (*dict*) – keyword arguments.

Supported Platforms:

`Ascend` `GPU`

query()

Checks if all the work submitted has been completed.

Returns

A boolean indicating if all kernels in this stream are completed.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1024, 2048]), ms.float32)
>>> b = Tensor(np.ones([2048, 4096]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
>>> s1.synchronize()
>>> assert s1.query()
```

record_event(*event=None*)

Records an event.

Parameters

event (*Event, optional*) – event to record. If not given, a new one will be allocated. Default is `None`.

Returns

Event, recorded event. If this argument is `None`, a new one will be allocated. Default is `None`.

Raises

TypeError – If 'event' is neither a *mindspore.runtime.Event* nor a `None`.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([3, 3]), ms.float32)
>>> b = Tensor(np.ones([3, 3]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = a + b
...     event = s1.record_event()
...     d = a * b
>>> cur_stream = ms.runtime.current_stream()
>>> cur_stream.wait_event(event)
>>> e = c + 3
>>> print(e)
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

synchronize()

Wait for all the kernels in this stream to complete.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1024, 2048]), ms.float32)
>>> b = Tensor(np.ones([2048, 4096]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
>>> s1.synchronize()
>>> assert s1.query()
```

wait_event(*event*)

Makes all future work submitted to the stream wait for an event.

Parameters

event (Event) – an event to wait for.

Raises

TypeError – If 'event' is not a *mindspore.runtime.Event*.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([3, 3]), ms.float32)
>>> b = Tensor(np.ones([3, 3]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = a + b
...     event = s1.record_event()
...     d = a * b
>>> cur_stream = ms.runtime.current_stream()
>>> cur_stream.wait_event(event)
>>> e = c + 3
>>> print(e)
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

wait_stream (*stream*)

Synchronizes with another stream.

All future work submitted to this stream will wait until all kernels submitted to a given stream at the time of call complete.

Parameters

stream (Stream) – a stream to synchronize.

Raises

TypeError – If 'stream' is not a *mindspore.runtime.Stream*.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> s1 = ms.runtime.Stream()
>>> s2 = ms.runtime.Stream()
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([2, 2]), ms.float32)
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
>>> with ms.runtime.StreamCtx(s2):
...     s2.wait_stream(s1)
...     d = ops.matmul(c, b)
>>> ms.runtime.synchronize()
>>> print(d)
[[4. 4.]]
```

9.3.7 mindspore.runtime.StreamCtx

class mindspore.runtime.**StreamCtx**(*ctx_stream*)

Context-manager that selects a given stream.

All kernels queued within its context will be enqueued on a selected stream.

Parameters

ctx_stream (Stream) –selected stream. This manager is a no-op if it's None.

Raises

TypeError –If 'stream' is neither a *mindspore.runtime.Stream* nor a None.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1024, 2048]), ms.float32)
>>> b = Tensor(np.ones([2048, 4096]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
>>> ms.runtime.synchronize()
>>> assert s1.query()
```

9.4 Event

mindspore.runtime.Event	Wrapper around a device event.

9.4.1 mindspore.runtime.Event

class mindspore.runtime.**Event**(*enable_timing=False*, *blocking=False*)

Wrapper around a device event.

Device events are synchronization markers that can be used to monitor the device' s progress, to accurately measure timing, and to synchronize device streams.

The underlying device events are lazily initialized when the event is first recorded.

Parameters

- **enable_timing** (*bool, optional*) –indicates if the event should measure time (default: False)
- **blocking** (*bool, optional*) –if True, *wait* will be blocking (default: False)

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> start = ms.runtime.Event(enable_timing=True)
>>> end = ms.runtime.Event(enable_timing=True)
>>> s1 = ms.runtime.Stream()
>>> s2 = ms.runtime.Stream()
>>> a = Tensor(np.ones([2, 2]), ms.float32)
>>> b = Tensor(np.ones([2, 2]), ms.float32)
>>> c = Tensor(np.ones([2, 2]), ms.float32)
>>> with ms.runtime.StreamCtx(s1):
...     d = ops.matmul(a, b)
...     start.record()
>>> c += 2
>>> end.record()
>>> with ms.runtime.StreamCtx(s2):
...     start.synchronize()
...     end.synchronize()
...     e = c + d
>>> ms.runtime.synchronize()
>>> print(e)
[[5. 5.]
 [5. 5.]]
>>> elapsed_time = start.elapsed_time(end)
```

elapsed_time(*end_event*)

Returns the time elapsed in milliseconds after the event was recorded and before the end_event was recorded.

Parameters

end_event (Event) – end event.

Returns

float, the time elapsed in milliseconds.

Raises

TypeError – If 'end_event' is not a *mindspore.runtime.Event*.

query()

Checks if all work currently captured by event has completed.

Returns

A boolean indicating if all work currently captured by event has completed.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> a = Tensor(np.ones([1024, 2048]), ms.float32)
>>> b = Tensor(np.ones([2048, 4096]), ms.float32)
>>> s1 = ms.runtime.Stream()
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
...     ev = s1.record_event()
>>> s1.synchronize()
>>> assert ev.query()
```

record(*stream=None*)

Records the event in a given stream.

Uses *mindspore.runtime.current_stream()* if no *stream* is specified. The stream's device must match the event's device.

Parameters

stream (Stream, *optional*) –a stream to record. If this argument is None, current stream will be used. Default value: None.

Raises

TypeError –If 'stream' is neither a *mindspore.runtime.Stream* nor a None.

synchronize()

Waits for the event to complete.

Waits until the completion of all work currently captured in this event. This prevents the CPU thread from proceeding until the event completes.

wait(*stream=None*)

Makes all future work submitted to the given stream wait for this event.

Use *mindspore.runtime.current_stream()* if no *stream* is specified.

Parameters

stream (Stream, *optional*) –a stream to record. If this argument is None, current stream will be used. Default value: None.

Raises

TypeError –If 'stream' is neither a *mindspore.runtime.Stream* nor a None.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.set_device("Ascend", 0)
>>> event = ms.runtime.Event()
>>> s1 = ms.runtime.Stream()
>>> s2 = ms.runtime.Stream()
>>> a = Tensor(np.ones([2, 2]), ms.float32)
>>> b = Tensor(np.ones([2, 2]), ms.float32)
>>> with ms.runtime.StreamCtx(s1):
...     c = ops.matmul(a, b)
...     event.record()
>>> event.wait()
>>> d = c + 2
>>> ms.runtime.synchronize()
>>> print(d)
[[4. 4.]
 [4. 4.]]
```

CHAPTER

TEN

MINDSPORE.DEVICE_CONTEXT

The device_context encapsulates the interface for querying the number of devices with whether the currently specified backend is available.

10.1 Cpu Device Backend Management

mindspore.device_context.cpu.device_count	Returns compute-capable device count of CPU.
mindspore.device_context.cpu.is_available	Return whether cpu backend is available.
mindspore.device_context.cpu.op_tuning.threads_num	Set the threads number of CPU kernel used.

10.1.1 mindspore.device_context.cpu.device_count

mindspore.device_context.cpu.**device_count**()

Returns compute-capable device count of CPU.

Note:

- For CPU hardware, 1 is always returned.

Returns

int, The number of compute-capable CPU devices.

Examples

```
>>> import mindspore as ms
>>> print(ms.device_context.cpu.device_count())
1
```

10.1.2 mindspore.device_context.cpu.is_available

mindspore.device_context.cpu.**is_available**()

Return whether cpu backend is available.

Returns

Bool, whether the cpu backend is available for this MindSpore package.

Examples

```
>>> import mindspore as ms
>>> print(ms.device_context.cpu.is_available())
True
```

10.1.3 mindspore.device_context.cpu.op_tuning.threads_num

mindspore.device_context.cpu.op_tuning.**threads_num**(*num*)

Set the threads number of CPU kernel used.

The framework set the threads number of CPU kernel used are 25 by default.

Parameters

num (*int*) −The threads number of CPU kernel used.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 1)
>>> ms.device_context.cpu.op_tuning.threads_num(10)
```

10.2 Gpu Device Backend Management

mindspore.device_context.gpu.device_count	Return the number of GPUs available.
mindspore.device_context.gpu.is_available	Return a bool indicating if CUDA is currently available.
mindspore.device_context.gpu.op_precision.conv_allow_tf32	Whether to convert FP32 to TF32 for Conv operators.
mindspore.device_context.gpu.op_precision.matmul_allow_tf32	Whether to convert FP32 to TF32 for Matmul operators.
mindspore.device_context.gpu.op_tuning.conv_dgrad_algo	Specifies convolution data grad algorithm.
mindspore.device_context.gpu.op_tuning.conv_fprop_algo	Specifies convolution forward algorithm.
mindspore.device_context.gpu.op_tuning.conv_wgrad_algo	Specifies convolution filter grad algorithm.

10.2.1 mindspore.device_context.gpu.device_count

mindspore.device_context.gpu.**device_count**()

Return the number of GPUs available.

Returns

Bool, the number of GPUs available.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.device.device_count()
```

10.2.2 mindspore.device_context.gpu.is_available

mindspore.device_context.gpu.**is_available**()

Return a bool indicating if CUDA is currently available.

Returns

Bool, indicating if CUDA is currently available.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.device.is_available()
```

10.2.3 mindspore.device_context.gpu.op_precision.conv_allow_tf32

mindspore.device_context.gpu.op_precision.**conv_allow_tf32**(*value*)

Whether to convert FP32 to TF32 for Conv operators. For detailed information, please refer to CUBLAS_COMPUTE_32F_FAST_TF32.

Parameters

value (*bool*) – Whether to convert FP32 to HF32 for Conv operators. If not configured, the framework defaults to `True`.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.op_precision.conv_allow_tf32(False)
```

10.2.4 mindspore.device_context.gpu.op_precision.matmul_allow_tf32

mindspore.device_context.gpu.op_precision.**matmul_allow_tf32**(*value*)

Whether to convert FP32 to TF32 for Matmul operators. For detailed information, please refer to CUBLAS_COMPUTE_32F_FAST_TF32.

Parameters

value (*bool*) – Whether to convert FP32 to TF32 for Matmul operators. If not configured, the framework defaults to `False`.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.op_precision.matmul_allow_tf32(True)
```

10.2.5 mindspore.device_context.gpu.op_tuning.conv_dgrad_algo

mindspore.device_context.gpu.op_tuning.**conv_dgrad_algo**(*mode*)

Specifies convolution data grad algorithm. For detailed information, please refer to NVIDA cuDNN.

Parameters

mode (*str*) – convolution data grad algorithm. If not configured, the framework defaults to 'normal'. The value range is as follows:

- normal: Use the cuDNN's heuristic search algorithm, the appropriate convolution algorithm will be quickly selected based on the convolution shape and type. This parameter does not guarantee optimal performance.
- performance: Use the cuDNN's trial search algorithm, all convolution algorithms will be trial run based on the convolution shape and type, and the optimal algorithm will be selected. This parameter ensures optimal performance.
- algo_0: This algorithm expresses the convolution as a sum of matrix products without actually explicitly forming the matrix that holds the input tensor data. The sum is done using the atomic add operation, thus the results are non-deterministic.
- algo_1: This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data. The results are deterministic.
- fft: This algorithm uses a Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results. The results are deterministic.
- fft_tiling: This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than fft for large size images. The results are deterministic.
- winograd: This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results. The results are deterministic.
- winograd_nonfused: This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results. The results are deterministic.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.op_tuning.conv_dgrad_algo("performance")
```

10.2.6 mindspore.device_context.gpu.op_tuning.conv_fprop_algo

mindspore.device_context.gpu.op_tuning.**conv_fprop_algo**(*mode*)

Specifies convolution forward algorithm. For detailed information, please refer to NVIDA cuDNN about cudnnConvolutionForward.

Parameters

mode (*str*) – convolution forward algorithm. If not configured, the framework defaults to 'normal'. The value range is as follows:

- normal: Use the cuDNN's heuristic search algorithm, the appropriate convolution algorithm will be quickly selected based on the convolution shape and type. This parameter does not guarantee optimal performance.
- performance: Use the cuDNN's trial search algorithm, all convolution algorithms will be trial run based on the convolution shape and type, and the optimal algorithm will be selected. This parameter ensures optimal performance.
- implicit_gemm: This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data.
- precomp_gemm: This algorithm expresses convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.
- gemm: This algorithm expresses the convolution as an explicit matrix product. A significant memory workspace is needed to store the matrix that holds the input tensor data.
- direct: This algorithm expresses the convolution as a direct convolution, without implicitly or explicitly doing a matrix multiplication.
- fft: This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results.
- fft_tiling: This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than fft algorithm for large size images.
- winograd: This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.
- winograd_nonfused: This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.op_tuning.conv_fprop_algo("performance")
```

10.2.7 mindspore.device_context.gpu.op_tuning.conv_wgrad_algo

mindspore.device_context.gpu.op_tuning.**conv_wgrad_algo**(*mode*)

Specifies convolution filter grad algorithm. For detailed information, please refer to NVIDA cuDNN.

Parameters

mode (*str*) – convolution filter grad algorithm. If not configured, the framework defaults to 'normal'. The value range is as follows:

- normal: Use the cuDNN's heuristic search algorithm, the appropriate convolution algorithm will be quickly selected based on the convolution shape and type. This parameter does not guarantee optimal performance.
- performance: Use the cuDNN's trial search algorithm, all convolution algorithms will be trial run based on the convolution shape and type, and the optimal algorithm will be selected. This parameter ensures optimal performance.
- algo_0: This algorithm expresses the convolution as a sum of matrix products without actually explicitly forming the matrix that holds the input tensor data. The sum is done using the atomic add operation, thus the results are non-deterministic.
- algo_1: This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data. The results are deterministic.
- algo_3: This algorithm is similar to algo_0 but uses some small workspace to precompute some indices. The results are also non-deterministic.
- fft: This algorithm uses a Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results. The results are deterministic.
- fft_tiling: This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than fft for large size images. The results are deterministic.
- winograd_nonfused: This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results. The results are deterministic.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.gpu.op_tuning.conv_wgrad_algo("performance")
```

10.3 Ascend Device Backend Management

mindspore.device_context.ascend.device_count	Return compute-capable device count of Ascend.
mindspore.device_context.ascend.is_available	Returns whether ascend backend is available.
mindspore.device_context.ascend.op_precision.conv_allow_hf32	Whether to convert FP32 to HF32 for Conv operators.
mindspore.device_context.ascend.op_precision.matmul_allow_hf32	Whether to convert FP32 to HF32 for Matmul operators.
mindspore.device_context.ascend.op_precision.precision_mode	Configure mixed precision mode setting.
mindspore.device_context.ascend.op_precision.op_precision_mode	Path to config file of op precision mode.

continues on next page

Table 3 – continued from previous page

mindspore.device_context.ascend.op_debug.execute_timeout	Set the maximum duration of executing an operator in seconds.
mindspore.device_context.ascend.op_debug.debug_option	Enable debugging options for Ascend operators, default not enabled.
mindspore.device_context.ascend.op_debug.aclinit_config	Configure the configuration items for the aclInit interface.
mindspore.device_context.ascend.op_tuning.op_compile	Whether to select online compilation.The default settings by the framework are online compilation for static shape, and compiled operator binary files for dynamic shape.
mindspore.device_context.ascend.op_tuning.aoe_tune_mode	AOE tuning mode setting, which is not set by default.
mindspore.device_context.ascend.op_tuning.aoe_job_type	Set the parameters specific to Ascend Optimization Engine.It needs to be used in conjunction with mindspore.device_context.op_tuning.aoe_tune_mode(tune_mode).

10.3.1 mindspore.device_context.ascend.device_count

mindspore.device_context.ascend.**device_count**()

Return compute-capable device count of Ascend.

Returns

int, the number of compute-capable Ascend devices.

Examples

```
>>> import mindspore as ms
>>> print(ms.device_context.ascend.device_count())
8
```

10.3.2 mindspore.device_context.ascend.is_available

mindspore.device_context.ascend.**is_available**()

Returns whether ascend backend is available.

Returns

Bool, whether the ascend backend is available for this MindSpore package.

Examples

```
>>> import mindspore as ms
>>> print(ms.device_context.ascend.is_available())
True
```

10.3.3 mindspore.device_context.ascend.op_precision.conv_allow_hf32

mindspore.device_context.ascend.op_precision.**conv_allow_hf32**(*value*)

Whether to convert FP32 to HF32 for Conv operators. CANN enables FP32 to HF32 for Conv operators by default. For detailed information, please refer to Ascend community .

Note:

- This is an experimental prototype that is subject to change and/or deletion.

Parameters

value (*bool*) – Whether to convert FP32 to HF32 for Conv operators.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_precision.conv_allow_hf32(True)
```

10.3.4 mindspore.device_context.ascend.op_precision.matmul_allow_hf32

mindspore.device_context.ascend.op_precision.**matmul_allow_hf32**(*value*)

Whether to convert FP32 to HF32 for Matmul operators. CANN disables FP32 to HF32 for Matmul operators by default. For detailed information, please refer to Ascend community .

Note:

- This is an experimental prototype that is subject to change and/or deletion.

Parameters

value (*bool*) – Whether to convert FP32 to HF32 for Matmul operators

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_precision.matmul_allow_hf32(True)
```

10.3.5 mindspore.device_context.ascend.op_precision.precision_mode

mindspore.device_context.ascend.op_precision.**precision_mode**(*mode*)

Configure mixed precision mode setting. The framework set the configuration of Atlas training series products to "force_fp16" by default, and set the configuration for other products such as the Atlas A2 training series products to "must_keep_origin_dtype" by default. For detailed information, please refer to Ascend community .

Note:

- The default value of *precision_mode* is experimental parameter, may change in the future.

Parameters

mode (*str*) –The operator precision mode setting. The value range is as follows:

- force_fp16: When the operator supports both float16 and float32, directly choose float16.
- allow_fp32_to_fp16: For matrix-type operators, use float16. For vector-type operators, prioritize the original precision. If the operator in the network model supports float32, retain the original precision float32. If the operator in the network model does not support float32, directly reduce the precision to float16.
- allow_mix_precision: Automatic mixed precision, for all operators in the network, according to the built-in optimization strategy, automatically reduce the precision of some operators to float16 or bfloat16.
- must_keep_origin_dtype: Maintain the original precision.
- force_fp32: When the input of the matrix calculation operator is float16, and the output supports both float16 and float32, force the output to be converted to float32.
- allow_fp32_to_bf16: For matrix-type operators, use bfloat16. For vector-type operators, prioritize the original precision. If the operator in the network model supports float32, retain the original precision float32. If the operator in the network model does not support float32, directly reduce the precision to bfloat16.
- allow_mix_precision_fp16: Automatic mixed precision, for all operators in the network, according to the built-in optimization strategy, automatically reduce the precision of some operators to float16.
- allow_mix_precision_bf16: Automatic mixed precision, for all operators in the network, according to the built-in optimization strategy, automatically reduce the precision of some operators to bfloat16.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_precision.precision_mode("force_fp16")
```

10.3.6 mindspore.device_context.ascend.op_precision.op_precision_mode

mindspore.device_context.ascend.op_precision.**op_precision_mode**(*path*)

Path to config file of op precision mode. For detailed information, please refer to Ascend community .

Parameters

path (*str*) –Directory of the configuration file (.ini format) for setting the operator precision mode. The directory can contain letters, digits, underscores (_), hyphens (-), and periods (.).

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_precision.op_precision_mode("./op_precision_config_file")
```

10.3.7 mindspore.device_context.ascend.op_debug.execute_timeout

mindspore.device_context.ascend.op_debug.**execute_timeout**(*op_timeout*)

Set the maximum duration of executing an operator in seconds. The framework operator execution timeout time is 900 by default. please refer to Ascend Community document about aclrtSetOpExecuteTimeOut.

Parameters

op_timeout (*int*) −Set the maximum duration of executing an operator in seconds. If the execution time exceeds this value, system will terminate the task. 0 means endless wait. The defaults for AI Core and AI CPU operators vary on different hardware.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_debug.execute_timeout(100)
```

10.3.8 mindspore.device_context.ascend.op_debug.debug_option

mindspore.device_context.ascend.op_debug.**debug_option**(*option_value*)

Enable debugging options for Ascend operators, default not enabled.

Parameters

option_value (*str*) −Ascend operators debugging configuration. Currently, only memory access violation detection is supported. The value currently only supports being set to "oom".

- "oom": When there is a memory out of bounds during the execution of an operator, AscendCL will return an error code of EZ9999.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_debug.debug_option("oom")
```

10.3.9 mindspore.device_context.ascend.op_debug.aclinit_config

mindspore.device_context.ascend.op_debug.**aclinit_config**(*config*)

Configure the configuration items for the aclInit interface. please refer to Ascend Community document about aclInit..

Parameters

config (*dict*) −When initializing AscendCL, you can enable or configure the following features through this configuration interface.

- "max_opqueue_num": When executing using the single-operator model method, to save memory and balance the performance of calls, you can configure the maximum length of the single-operator model mapping queue through the max_opqueue_num parameter. If the length reaches the maximum, the system will first delete the mapping information that has not been used for a long time and the cached single-operator model, and then load the latest mapping information and the corresponding single-operator model. If the maximum length of the mapping queue is not configured, the default maximum length is 20,000.

- "err_msg_mode": This parameter is used to control the level at which error information is retrieved, either by process or by thread. The default level is by process. "0" indicating that error information is retrieved by thread. "1" is the default value, indicates that error information is retrieved by process.
- "dump": This parameter is used to enable exception dump for Ascend operators. The value can be set to {"dump_scene": "lite_exception"}, {"dump_scene": "lite_exception:disable"}. {"dump_scene": "lite_exception"} indicates that the exception dump is enabled. {"dump_scene": "lite_exception:disable"} indicates that the exception dump is disabled. {"dump_scene": "lite_exception"} is the default value, indicates that the exception dump is enabled.

Examples

```
>>> import mindspore as ms
>>> ms.set_device("Ascend", 0)
>>> ms.device_context.ascend.op_debug.aclinit_config({"max_opqueue_num": "20000", "err_msg_
↪mode": "1",
...                                                   "dump": {"dump_scene": "lite_exception
↪"}})
```

10.3.10 mindspore.device_context.ascend.op_tuning.op_compile

mindspore.device_context.ascend.op_tuning.**op_compile**(*value*)

Whether to select online compilation.The default settings by the framework are online compilation for static shape, and compiled operator binary files for dynamic shape. The default settings may change in the future. For detailed information, please refer to Ascend community .

Parameters

value (*bool*) – Whether to select online compilation or not.

- True: online compilation is prioritized.
- False: compiled operator binary files are prioritized to improve compilation performance.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_tuning.op_compile(True)
```

10.3.11 mindspore.device_context.ascend.op_tuning.aoe_tune_mode

mindspore.device_context.ascend.op_tuning.**aoe_tune_mode**(*tune_mode*)

AOE tuning mode setting, which is not set by default.

Parameters

tune_mode (*str*) – AOE tuning mode setting.

- "online": the online tuning function is turned on.
- "offline": ge graph will be saved for offline tuning.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_tuning.aoe_tune_mode("online")
```

10.3.12 mindspore.device_context.ascend.op_tuning.aoe_job_type

mindspore.device_context.ascend.op_tuning.**aoe_job_type**(*config*)

Set the parameters specific to Ascend Optimization Engine.It needs to be used in conjunction with mindspore.device_context.op_tuning.aoe_tune_mode(tune_mode). The framework set to "2" by default.

Parameters

config (*str*) – Choose the tuning type.

- "1": Set to subgraph tuning.
- "2": Set to operator tuning.

Examples

```
>>> import mindspore as ms
>>> ms.device_context.ascend.op_tuning.aoe_job_type("1")
```

CHAPTER

ELEVEN

MINDSPORE.COMMUNICATION

Collective communication interface.

Note that the APIs in the following list need to preset communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

mindspore.communication.GlobalComm	World communication information.
mindspore.communication.init	Initialize distributed backends required by communication services, e.g.
mindspore.communication.release	Release distributed resource.
mindspore.communication.create_group	Create a user collective communication group.
mindspore.communication.destroy_group	Destroy the user collective communication group.
mindspore.communication.get_comm_name	Get the communicator name of the specified collective communication group.
mindspore.communication.get_group_size	Get the rank size of the specified collective communication group.
mindspore.communication.get_group_rank_from_world_rank	Get the rank ID in the specified user communication group corresponding to the rank ID in the world communication group.
mindspore.communication.get_local_rank	Gets local rank ID for current device in specified collective communication group.
mindspore.communication.get_local_rank_size	Gets local rank size of the specified collective communication group.
mindspore.communication.get_process_group_ranks	Gets the ranks of the specific group and returns the process ranks in the communication group as a list.
mindspore.communication.get_rank	Get the rank ID for the current device in the specified collective communication group.
mindspore.communication.get_world_rank_from_group_rank	Get the rank ID in the world communication group corresponding to the rank ID in the specified user communication group.
mindspore.communication.HCCL_WORLD_COMM_GROUP	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
mindspore.communication.NCCL_WORLD_COMM_GROUP	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
mindspore.communication.MCCL_WORLD_COMM_GROUP	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

11.1 mindspore.communication.GlobalComm

class mindspore.communication.**GlobalComm**

World communication information. The GlobalComm is a global class. The members contain:

- BACKEND : The communication library used, using "hccl" / "nccl" / "mccl" . "hccl" means Huawei Collective Communication Library(HCCL), "nccl" means NVIDIA Collective Communication Library(NCCL), "mccl" means MindSpore Collective Communication Library(MCCL).
- WORLD_COMM_GROUP : Global communication domain, using "hccl_world_group" / "nccl_world_group" / "mccl_world_group" .

11.2 mindspore.communication.init

mindspore.communication.**init**(*backend_name=None*)

Initialize distributed backends required by communication services, e.g. "hccl" / "nccl" / "mccl". It is usually used in distributed parallel scenarios and set before using communication services.

Note:

- The full name of "hccl" is Huawei Collective Communication Library(HCCL).
- The full name of "nccl" is NVIDIA Collective Communication Library(NCCL).
- The full name of "mccl" is MindSpore Collective Communication Library(MCCL).
- In Ascend hardware platforms, init() should be set before the definition of any Tensor and Parameter, and the instantiation and execution of any operation and net.

Parameters

backend_name (*str*) –Backend, using "hccl" / "nccl" / "mccl". "hccl" should be used for Ascend hardware platforms, "nccl" for GPU hardware platforms and "mccl" for CPU hardware platforms. If not set, inference is automatically made based on the hardware platform type (device_target). Default: None .

Raises

- **TypeError** –If *backend_name* is not a string.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails, or the environment variables RANK_ID/MINDSPORE_HCCL_CONFIG_PATH have not been exported when backend is HCCL.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication import init
>>> init()
```

11.3 mindspore.communication.release

mindspore.communication.**release**()
Release distributed resource. e.g. HCCL/NCCL/MCCL.

Note: This method should be used after init(). If not, resource will be released when program ends.

Raises

RuntimeError – If failed to release distributed resource.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication import init, release
>>> init()
>>> release()
```

11.4 mindspore.communication.create_group

mindspore.communication.**create_group**(*group*, *rank_ids*)
Create a user collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. The size of rank_ids should be larger than 1, rank_ids should not have duplicate data. This method should be used after init(). Only support global single communication group in PyNative mode if you do not start with mpirun.

Parameters

- **group** (*str*) −The name of the communication group to be created.
- **rank_ids** (*list*) −A list of device IDs.

Raises

- **TypeError** −If group is not a string or *rank_ids* is not a list.
- **ValueError** −If *rank_ids* size is not larger than 1, or *rank_ids* has duplicate data, or backend is invalid.
- **RuntimeError** −If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore import set_context
>>> from mindspore import ops
>>> from mindspore.communication import init, create_group, get_rank
>>> set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> group = "0-7"
>>> rank_ids = [0,7]
>>> if get_rank() in rank_ids:
...     create_group(group, rank_ids)
...     allreduce = ops.AllReduce(group)
```

11.5 mindspore.communication.destroy_group

mindspore.communication.**destroy_group**(*group*)

Destroy the user collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. The parameter group should not be "hccl_world_group". This method should be used after init().

Parameters

group (*str*) −The communication group to destroy, the group should be created by create_group.

Raises

- **TypeError** −If group is not a string.

- **ValueError** –If group is "hccl_world_group" or backend is invalid.
- **RuntimeError** –If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore import set_context
>>> from mindspore import ops
>>> from mindspore.communication import init, create_group, destroy_group, get_rank
>>> set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> group = "0-2"
>>> rank_ids = [0,2]
>>> if get_rank() in rank_ids:
...     create_group(group, rank_ids)
...     destroy_group(group)
```

11.6 mindspore.communication.get_comm_name

mindspore.communication.**get_comm_name**(*group=GlobalComm.WORLD_COMM_GROUP*)

Get the communicator name of the specified collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. This method should be used after init().

Parameters

group (*str*) –The communication group to work on. Normally, the group should be created by create_group, otherwise, using the default group. Default: `GlobalComm.WORLD_COMM_GROUP` .

Returns

string, the inner communicator name of the group.

Raises

- **TypeError** –If group is not a string.
- **ValueError** –If backend is invalid.
- **RuntimeError** –If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.communication import init, create_group, get_rank, get_comm_name
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> world_group_comm_name = get_comm_name()
>>> group = "0-7"
>>> rank_ids = [0,7]
>>> if get_rank() in rank_ids:
...     create_group(group, rank_ids)
...     customizd_group_comm_name = get_comm_name(group)
...     print("comm_name of customizd group is ", customizd_group_comm_name)
>>> print("comm_name of world group is: ", world_group_comm_name)
comm_name of customizd group is: 11.22.33.44%eth0_60000_0_0123456789101112
comm_name of world group is: 11.22.33.44%eth0_60000_0_1211109876543210
```

11.7 mindspore.communication.get_group_size

mindspore.communication.**get_group_size**(*group=GlobalComm.WORLD_COMM_GROUP*)

Get the rank size of the specified collective communication group.

Note: This method should be used after init().

Parameters

group (*str*) – The communication group to work on. Normally, the group should be created by create_group, otherwise, using the default group. Default: `GlobalComm.WORLD_COMM_GROUP`.

Returns

int, the rank size of the group.

Raises

- **TypeError** – If group is not a string.
- **ValueError** – If backend is invalid.
- **RuntimeError** – If HCCL/NCCL/MCCL is not available.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.communication import init, get_group_size
>>> init()
>>> group_size = get_group_size()
>>> print("group_size is: ", group_size)
group_size is: 8
```

11.8 mindspore.communication.get_group_rank_from_world_rank

mindspore.communication.**get_group_rank_from_world_rank**(*world_rank_id*, *group*)

Get the rank ID in the specified user communication group corresponding to the rank ID in the world communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. The parameter group should not be "hccl_world_group". This method should be used after init().

Parameters

- **world_rank_id** (*int*) –A rank ID in the world communication group.
- **group** (*str*) –The communication group to work on. The group is created by create_group.

Returns

int, the rank ID in the user communication group.

Raises

- **TypeError** –If world_rank_id is not an integer or the group is not a string.
- **ValueError** –If group is 'hccl_world_group' or backend is invalid.
- **RuntimeError** –If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore import set_context
>>> from mindspore.communication import init, create_group, get_group_rank_from_world_rank,␣
↪get_rank
>>> set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> group = "0-4"
>>> rank_ids = [0,4]
>>> if get_rank() in rank_ids:
...     create_group(group, rank_ids)
...     group_rank_id = get_group_rank_from_world_rank(4, group)
...     print("group_rank_id is: ", group_rank_id)
group_rank_id is: 1
```

11.9 mindspore.communication.get_local_rank

mindspore.communication.**get_local_rank**(*group=GlobalComm.WORLD_COMM_GROUP*)

Gets local rank ID for current device in specified collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. This method should be used after init().

Parameters

group (*str*) – The communication group to work on. Normally, the group should be created by create_group, otherwise, using the default group. Default: GlobalComm.WORLD_COMM_GROUP.

Returns

int, the local rank ID of the calling process within the group.

Raises

- **TypeError** – If group is not a string.
- **ValueError** – If backend is invalid.
- **RuntimeError** – If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.communication import init, get_rank, get_local_rank
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> world_rank = get_rank()
>>> local_rank = get_local_rank()
>>> print("local_rank is: {}, world_rank is {}".format(local_rank, world_rank))
local_rank is: 1, world_rank is 9
```

11.10 mindspore.communication.get_local_rank_size

mindspore.communication.**get_local_rank_size**(*group=GlobalComm.WORLD_COMM_GROUP*)

Gets local rank size of the specified collective communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. This method should be used after init().

Parameters

group (*str*) – The communication group to work on. The group is created by create_group or the default world communication group. Default: `GlobalComm.WORLD_COMM_GROUP` .

Returns

int, the local rank size where the calling process is within the group.

Raises

- **TypeError** – If group is not a string.
- **ValueError** – If backend is invalid.
- **RuntimeError** – If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore.communication import init, get_local_rank_size
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> local_rank_size = get_local_rank_size()
>>> print("local_rank_size is: ", local_rank_size)
local_rank_size is: 8
```

11.11 mindspore.communication.get_process_group_ranks

mindspore.communication.**get_process_group_ranks**(*group=GlobalComm.WORLD_COMM_GROUP*)

Gets the ranks of the specific group and returns the process ranks in the communication group as a list.

Parameters

group (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Returns

List (List[int]), List of process ranks in the specified communication group.

Raises

- **TypeError** – If the *group* is not a str.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import numpy as np
>>> from mindspore.communication import init, get_process_group_ranks
>>>
```

(continues on next page)

(continued from previous page)

```
>>> init()
>>> output = get_process_group_ranks()
>>> print(output)
[0, 1, 2, 3]
```

11.12 mindspore.communication.get_rank

mindspore.communication.**get_rank**(*group=GlobalComm.WORLD_COMM_GROUP*)

Get the rank ID for the current device in the specified collective communication group.

Note: This method should be used after init().

Parameters

group (*str*) – The communication group to work on. Normally, the group should be created by create_group, otherwise, using the default group. Default: `GlobalComm.WORLD_COMM_GROUP` .

Returns

int, the rank ID of the calling process within the group.

Raises

- **TypeError** – If group is not a string.
- **ValueError** – If backend is invalid.
- **RuntimeError** – If HCCL/NCCL/MCCL is not available.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> from mindspore.communication import init, get_rank
>>> init()
>>> rank_id = get_rank()
>>> print(rank_id)
>>> # the result is the rank_id in world_group
```

11.13 mindspore.communication.get_world_rank_from_group_rank

mindspore.communication.**get_world_rank_from_group_rank**(*group*, *group_rank_id*)

Get the rank ID in the world communication group corresponding to the rank ID in the specified user communication group.

Note: This method isn't supported in GPU and CPU versions of MindSpore. The parameter group should not be "hccl_world_group". This method should be used after init().

Parameters

- **group** (*str*) −The communication group to work on. The group is created by create_group.
- **group_rank_id** (*int*) −A rank ID in the communication group.

Returns

int, the rank ID in world communication group.

Raises

- **TypeError** −If *group_rank_id* is not an integer or the group is not a string.
- **ValueError** −If group is 'hccl_world_group' or backend is invalid.
- **RuntimeError** −If HCCL is not available or MindSpore is GPU/CPU version.

Supported Platforms:

Ascend GPU CPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

```
>>> import mindspore as ms
>>> from mindspore import set_context
>>> from mindspore.communication import init, create_group, get_world_rank_from_group_rank,␣
 ↪get_rank
>>> set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>> init()
>>> group = "0-4"
>>> rank_ids = [0,4]
>>> if get_rank() in rank_ids:
...     create_group(group, rank_ids)
...     world_rank_id = get_world_rank_from_group_rank(group, 1)
...     print("world_rank_id is: ", world_rank_id)
world_rank_id is: 4
```

11.14 mindspore.communication.HCCL_WORLD_COMM_GROUP

mindspore.communication.**HCCL_WORLD_COMM_GROUP**

The string of "hccl_world_group" referring to the default communication group created by HCCL. On the Ascend hardware platforms, the string is equivalent to `GlobalComm.WORLD_COMM_GROUP` after the communication service is initialized. It is recommended to use `GlobalComm.WORLD_COMM_GROUP` to obtain the current global communication group.

11.15 mindspore.communication.NCCL_WORLD_COMM_GROUP

mindspore.communication.**NCCL_WORLD_COMM_GROUP**

The string of "nccl_world_group" referring to the default communication group created by NCCL. On the GPU hardware platforms, the string is equivalent to `GlobalComm.WORLD_COMM_GROUP` after the communication service is initialized. It is recommended to use `GlobalComm.WORLD_COMM_GROUP` to obtain the current global communication group.

11.16 mindspore.communication.MCCL_WORLD_COMM_GROUP

mindspore.communication.**MCCL_WORLD_COMM_GROUP**

The string of "mccl_world_group" referring to the default communication group created by MCCL. On the CPU hardware platforms, the string is equivalent to `GlobalComm.WORLD_COMM_GROUP` after the communication service is initialized. It is recommended to use `GlobalComm.WORLD_COMM_GROUP` to obtain the current global communication group.

11.17 mindspore.communication.comm_func

Collection communication functional interface

mindspore.communication.comm_func.all_gather_into_tensor	Gathers tensors from the specified communication group and returns the tensor which is all gathered.
mindspore.communication.comm_func.all_reduce	Reduce tensors across all devices in such a way that all deviceswill get the same final result, returns the tensor which is all reduced.
mindspore.communication.comm_func.all_to_all_single_with_output_shape	Based on the slice size of the user input, the input *tensor* is sliced and sent to other devices and receives the sliced chunks from the other devices, which are then merged into an output Tensor.
mindspore.communication.comm_func.all_to_all_with_output_shape	scatter and gather list of tensor to/from all rank according to input/output tensor list.
mindspore.communication.comm_func.barrier	Synchronizes all processes in the specified group.
mindspore.communication.comm_func.batch_isend_irecv	Batch send and recv tensors asynchronously.
mindspore.communication.comm_func.broadcast	Broadcasts the tensor to the whole group.
mindspore.communication.comm_func.gather_into_tensor	Gathers tensors from the specified communication group.
mindspore.communication.comm_func.irecv	Receive tensors from src asynchronously.
mindspore.communication.comm_func.isend	Send tensors to the specified dest_rank asynchronously.
mindspore.communication.comm_func.recv	Receive tensors from src.
mindspore.communication.comm_func.send	Send tensors to the specified dest_rank.

continues on next page

Table 2 – continued from previous page

mindspore.communication.comm_func.P2POp	Object for *batch_isend_irecv* input, to store information of "isend" and "irecv".
mindspore.communication.comm_func.reduce	Reduces tensors across the processes in the specified communication group, sends the result to the target dst(global rank), and returns the tensor which is sent to the target process.
mindspore.communication.comm_func.reduce_scatter_tensor	Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.
mindspore.communication.comm_func.scatter_tensor	Scatter tensor evently across the processes in the specified communication group.

11.17.1 mindspore.communication.comm_func.all_gather_into_tensor

mindspore.communication.comm_func.**all_gather_into_tensor**(*tensor*, *group=GlobalComm.WORLD_COMM_GROUP*, *async_op=False*)

Gathers tensors from the specified communication group and returns the tensor which is all gathered.

Note:

- The tensors must have the same shape and format in all processes of the collection.

Parameters

- **tensor** (Tensor) – The input tensor to be all gathered into tensor. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **group** (*str, optional*) – The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP , which means "hccl_world_group" in Ascend, and "nccl_world_group" in GPU.
- **async_op** (*bool, optional*) – Whether this operator should be an async operator. Default: False .

Returns

Tuple(Tensor, CommHandle), if the number of devices in the group is N, then the shape of output tensor is $(N, x_1, x_2, ..., x_R)$. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If the type of the first input parameter is not Tensor, or *group* is not a str.
- **ValueError** – If the local rank id of the calling process in the group is larger than the group's rank size.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> input_tensor = ms.Tensor(np.ones([2, 8]).astype(np.float32))
>>> output, _ = comm.comm_func.all_gather_into_tensor(input_tensor)
>>> print(output)
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

11.17.2 mindspore.communication.comm_func.all_reduce

mindspore.communication.comm_func.**all_reduce**(*tensor*, *op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*, *async_op=False*)

Reduce tensors across all devices in such a way that all deviceswill get the same final result, returns the tensor which is all reduced.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **tensor** (Tensor) –The input tensor to be all reduced. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like sum, prod, max, and min. On the CPU, only 'sum' is supported. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False`.

Returns

Tuple(Tensor, CommHandle), the output tensor has the same shape of the input, i.e., $(x_1, x_2, ..., x_R)$. The contents depend on the specified operation. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> input_tensor = ms.Tensor(np.ones([2, 8]).astype(np.float32))
>>> output, _ = comm.comm_func.all_reduce(input_tensor)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

11.17.3 mindspore.communication.comm_func.all_to_all_single_with_output_shape

mindspore.communication.comm_func.**all_to_all_single_with_output_shape**(*output_shape*, *tensor*, *output_split_sizes=None*, *input_split_sizes=None*, *group=None*, *async_op=False*)

Based on the slice size of the user input, the input *tensor* is sliced and sent to other devices and receives the sliced chunks from the other devices, which are then merged into an output Tensor.

Note: 'output_shape' and 'tensor' shape should be match across ranks. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **output_shape** (*Union(Tensor, Tuple(int))*) –shape to indicate the shape of tensor gathered concatenated from remote rank.
- **tensor** (Tensor) –tensor to be scattered to remote rank.
- **output_split_sizes** (*Union(Tuple(int), List(int))*) –output split size at dim 0. If set to None, it means equally split by `world_size`. Default: None.
- **input_split_sizes** (*Union(Tuple(int), List(int))*) –input split size at dim 0. If set to None, it means equally split by `world_size`. Default: None.
- **group** (*str, optional*) –The communication group to work on. Default: None, which means "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False` .

Returns

Tuple(Tensor, CommHandle), the output tensor is gathered concatenated from remote ranks. If the numel of tensor gathered from remote is zero, it will return a Tensor with shape *()*, and value has no actual meanning. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If *tensor* is not tensor.
- **TypeError** – If *output_shape* is not tuple or tensors.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> rank = comm.get_rank()
>>> input = ms.Tensor([0, 1]) + rank * 2
>>> output_shape = (2,)
>>> result, _ = comm.comm_func.all_to_all_single_with_output_shape(output_shape, input)
>>> print(result)
rank 0:
[ 0.  2.]
rank 1:
[ 1.  3.]
```

11.17.4 mindspore.communication.comm_func.all_to_all_with_output_shape

mindspore.communication.comm_func.**all_to_all_with_output_shape**(*output_shape_list*, *input_tensor_list*, *group=None*, *async_op=False*)

scatter and gather list of tensor to/from all rank according to input/output tensor list.

Note: tensor shape in *output_shape_list* and *input_tensor_list* should be match across ranks. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **output_shape_list** (*Union[Tuple(Tensor), List(Tensor), Tuple(Tuple(int))]*) – List of shape that indicate the gathered tensors shape from remote ranks.

- **input_tensor_list** (*Union[Tuple(Tensor), List(Tensor)]*) – List of tensors to scatter to the remote rank.
- **group** (*str, optional*) – The communication group to work on. Default: None, which means "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **async_op** (*bool, optional*) – Whether this operator should be an async operator. Default: `False`.

Returns

Tuple(Tuple(Tensor), CommHandle), the tensors is gathered from remote ranks. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** – If *input_tensor_list* is not list of tensors.
- **TypeError** – If *output_shape_list* is not list of tuple or tensors.
- **TypeError** – If tensors in *input_tensor_list* are not the same type.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> this_rank = comm.get_rank()
>>> if this_rank == 0:
...     send_tensor_list = [ms.Tensor(1.), ms.Tensor([[2, 3], [4, 5.]])]
...     recv_tensor_list = [(), (2,)]
>>> if this_rank == 1:
...     send_tensor_list = [ms.Tensor([2, 2.]), ms.Tensor([4, 5, 6, 7.])]
...     recv_tensor_list = [(2, 2), (4,)]
>>> output, _ = comm.comm_func.all_to_all_with_output_shape(recv_tensor_list, send_tensor_
↪list)
>>> print(output)
rank 0:
(Tensor(shape=[], dtype=Float32, value= 1),
 Tensor(shape=[2], dtype=Float32, value= [2.00000000e+00, 2.00000000e+00]))
rank 1:
(Tensor(shape=[2, 2], dtype=Float32, value=
[[2.00000000e+00, 3.00000000e+00],
 [4.00000000e+00, 5.00000000e+00]]),
 Tensor(shape=[4], dtype=Float32, value=[4.00000000e+00, 5.00000000e+00, 6.00000000e+00, 7.
↪00000000e+00]))
```

11.17.5 mindspore.communication.comm_func.barrier

mindspore.communication.comm_func.**barrier**(*group=GlobalComm.WORLD_COMM_GROUP*)

Synchronizes all processes in the specified group. Once the process call this operation, it will be blocked until all processes call this operation. After all processes finish calling the operations, the blocked processes will be woken and continue their task.

Parameters

group (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.

Raises

RuntimeError – If backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> # Launch 2 processes.
>>> comm.init()
>>> comm.comm_func.barrier()
>>> print("barrier finish!")
barrier finish!
```

Tutorial Examples:

- Distributed Set Communication Primitives - Barrier

11.17.6 mindspore.communication.comm_func.batch_isend_irecv

mindspore.communication.comm_func.**batch_isend_irecv**(*p2p_op_list*)

Batch send and recv tensors asynchronously.

Note:

- The 'isend' and 'irecv' of *P2POp* in *p2p_op_list* between ranks need to match each other.
- *P2POp* in *p2p_op_list* can only use the same communication group.
- *tag* of *P2POp* in *p2p_op_list* is not support yet.
- *tensor* of *P2POp* in *p2p_op_list* will not be modified by result inplace.
- Only support PyNative mode, Graph mode is not currently supported.

Parameters

p2p_op_list (P2POp) – list contains *P2POp*. *P2POp* is type of *mindspore.communication.comm_func.P2POp*

Returns

tuple(Tensor). Output tensors is corresponding to *p2p_op_list*. At *P2POp* with 'isend' position, output tensor is a fake tensor with scalar, which has no meaning. At *P2POp* with 'irecv' position, output tensor is a tensor received from remote device.

Raises

TypeError – If *p2p_op_list* are not all type of *P2POp*.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> this_rank = comm.get_rank()
>>> world_size = comm.get_group_size()
>>> next_rank = (this_rank + 1) % world_size
>>> prev_rank = (this_rank + world_size - 1) % world_size
>>>
>>> send_tensor = ms.Tensor(this_rank + 1, dtype=ms.float32)
>>> recv_tensor = ms.Tensor(0., dtype=ms.float32)
>>>
>>> send_op = comm.comm_func.P2POp('isend', send_tensor, next_rank)
>>> recv_op = comm.comm_func.P2POp('irecv', recv_tensor, prev_rank)
>>>
>>> p2p_op_list = [send_op, recv_op]
>>> output = comm.comm_func.batch_isend_irecv(p2p_op_list)
>>> print(output)
rank 0:
(Tensor(shape=[], dtype=Float32, value= 0), Tensor(shape=[], dtype=Float32, value= 2))
rank 1:
(Tensor(shape=[], dtype=Float32, value= 0), Tensor(shape=[], dtype=Float32, value= 1))
```

11.17.7 mindspore.communication.comm_func.broadcast

mindspore.communication.comm_func.**broadcast**(*tensor*, *src=0*, *group=GlobalComm.WORLD_COMM_GROUP*)

Broadcasts the tensor to the whole group.

Note: The tensors must have the same shape and format in all processes of the collection. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The tensor to be broadcasted. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **src** (*int, optional*) –Specifies the rank(global rank) of the process that broadcast the tensor. And only process *src* will broadcast the tensor.
- **group** (*str, optional*) –The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP.

Returns

Tensor, tensor has the same shape as input tensor $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** –If src is not an integer or group is not a string.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend GPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> # Launch 2 processes.
>>>
>>> comm.init()
>>> data = ms.Tensor(np.arange(8).reshape([2, 4]).astype(np.float32))
>>> out = comm.comm_func.broadcast(tensor=data, src=0)
>>> print(out)
[[0. 1. 2. 3.]
 [4. 5. 6. 7.]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - Broadcast

11.17.8 mindspore.communication.comm_func.gather_into_tensor

mindspore.communication.comm_func.**gather_into_tensor**(*tensor*, *dst=0*, *group=GlobalComm.WORLD_COMM_GROUP*)

Gathers tensors from the specified communication group. The operation will gather the tensor from processes according to dimension 0.

Note: Only the tensor in process *dst* (global rank) will keep the gathered tensor. The other process will keep a tensor with shape [1], which has no mathematical meaning. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The tensor to be gathered. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **dst** (*int, optional*) –Specifies the rank(global rank) of the process that receive the tensor. And only process *dst* will receive the gathered tensor. Default: 0.
- **group** (*str, optional*) –The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP.

Returns

Tensor, the shape of output is $(\sum x_1, x_2, ..., x_R)$. The dimension 0 of data is equal to sum of the dimension of input tensor, and the other dimension keep the same.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> # Launch 2 processes.
>>>
>>> comm.init()
>>> input = ms.Tensor(np.arange(4).reshape([2, 2]).astype(np.float32))
```

(continues on next page)

(continued from previous page)

```
>>> output = comm.comm_func.gather_into_tensor(tensor=input, dst=0)
>>> print(output)
Process with rank 0: [[0. 1.],
                      [2. 3.],
                      [0. 1.],
                      [2. 3.]]
Process with rank 1: [0]
```

11.17.9 mindspore.communication.comm_func.irecv

mindspore.communication.comm_func.**irecv**(*tensor*, *src=0*, *group=GlobalComm.WORLD_COMM_GROUP*, *tag=0*)

Receive tensors from src asynchronously.

Note: Send and Receive must be used in combination and have same tag. The shape and dtype of input *tensor* is used to receive tensor, but the value of input *tensor* would not take effect. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) −The shape of tensor is $(x_1, x_2, ..., x_R)$. The shape and dtype of this tensor is used to receive tensor, but the value of input *tensor* would not take effect.
- **src** (*int, optional*) −A required integer identifying the source rank(global rank). Default: 0.
- **group** (*str, optional*) −The communication group to work on. Default: "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **tag** (*int, optional*) −A required integer identifying the send/recv message tag. The message will be received by the Send op with the same "tag". Default: 0.

Returns

Tuple(Tensor, CommHandle), the shape of output is $(x_1, x_2, ..., x_R)$. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** −If *src* is not an int or *group* is not a str.
- **ValueError** −If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore.communication.comm_func import isend, irecv
>>> from mindspore.communication import get_rank, get_group_size
>>>
>>> np.random.seed(1)
>>> init()
>>> rank = get_rank()
>>> size = get_group_size()
>>> x = np.ones([2, 2]).astype(np.float32) * 0.01 * (rank + 1)
>>> x2 = np.ones([2, 2]).astype(np.float32)
>>>
>>>
>>> if rank < size / 2:
...     _x = ms.Tensor(x)
...     isend(_x, rank + size // 2)
... else:
...     _x2 = ms.Tensor(x2)
...     output, handle = irecv(_x2, rank - size // 2)
...     handle.wait()
...     print(output)
rank1:
[[0.01  0.01]
 [0.01  0.01]]
```

11.17.10 mindspore.communication.comm_func.isend

mindspore.communication.comm_func.**isend**(*tensor*, *dst=0*, *group=GlobalComm.WORLD_COMM_GROUP*, *tag=0*)

Send tensors to the specified dest_rank asynchronously.

Note: Send and Receive must be used in combination and have same tag. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **dst** (*int, optional*) –A required integer identifying the destination rank(global rank). Default: 0.
- **group** (*str, optional*) –The communication group to work on. Default: "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **tag** (*int, optional*) –A required integer identifying the send/recv message tag. The message will be received by the Receive op with the same "tag". Default: 0.

Returns

CommHandle, it is an async work handle.

Raises

- **TypeError** –*dst* is not an int or *group* is not a str。
- **ValueError** –If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore.communication.comm_func import isend, irecv
>>> from mindspore.communication import get_rank, get_group_size
>>>
>>> np.random.seed(1)
>>> init()
>>> rank = get_rank()
>>> size = get_group_size()
>>> x = np.ones([2, 2]).astype(np.float32) * 0.01 * (rank + 1)
>>> x2 = np.ones([2, 2]).astype(np.float32)
>>>
>>>
>>> if rank < size / 2:
...     _x = ms.Tensor(x)
...     isend(_x, rank + size // 2)
... else:
...     _x2 = ms.Tensor(x2)
...     output, handle = irecv(_x2, rank - size // 2)
...     handle.wait()
...     print(output)
rank1:
[[0.01  0.01]
 [0.01  0.01]]
```

11.17.11 mindspore.communication.comm_func.recv

mindspore.communication.comm_func.**recv**(*tensor*, *src=0*, *group=GlobalComm.WORLD_COMM_GROUP*, *tag=0*)

Receive tensors from src.

Note: Send and Receive must be used in combination and have same tag. The shape and dtype of input *tensor* is used to receive tensor, but the value of input *tensor* would not take effect. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The shape of tensor is $(x_1, x_2, ..., x_R)$. The shape and dtype of this tensor is used to receive tensor, but the value of input *tensor* would not take effect.
- **src** (*int, optional*) –A required integer identifying the source rank(global rank). Default: 0.

- **group** (*str, optional*) –The communication group to work on. Default: "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **tag** (*int, optional*) –A required integer identifying the send/recv message tag. The message will be received by the Send op with the same "tag". Default: 0.

Returns

Tensor, the shape of output is $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** –If *src* is not an int or *group* is not a str.
- **ValueError** –If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore.communication.comm_func import send, recv
>>> from mindspore.communication import get_rank, get_group_size
>>>
>>> np.random.seed(1)
>>> init()
>>> rank = get_rank()
>>> size = get_group_size()
>>> x = np.ones([2, 2]).astype(np.float32) * 0.01 * (rank + 1)
>>> x2 = np.ones([2, 2]).astype(np.float32)
>>>
>>>
>>> if rank < size / 2:
...     _x = ms.Tensor(x)
...     send(_x, rank + size // 2)
... else:
...     _x2 = ms.Tensor(x2)
...     output = recv(_x2, rank - size // 2)
...     print(output)
rank1:
[[0.01  0.01]
 [0.01  0.01]]
```

11.17.12 mindspore.communication.comm_func.send

mindspore.communication.comm_func.**send**(*tensor*, *dst=0*, *group=GlobalComm.WORLD_COMM_GROUP*, *tag=0*)

Send tensors to the specified dest_rank.

Note: Send and Receive must be used in combination and have same tag.

Parameters

- **tensor** (Tensor) – The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **dst** (*int, optional*) – A required integer identifying the destination rank(global rank). Default: 0.
- **group** (*str, optional*) – The communication group to work on. Default: "hccl_world_group" on Ascend, "nccl_world_group" on GPU.
- **tag** (*int, optional*) – A required integer identifying the send/recv message tag. The message will be received by the Receive op with the same "tag". Default: 0.

Raises

- **TypeError** – *dst* is not an int or *group* is not a str。
- **ValueError** – If the rank ID of the process is greater than the rank size of the communication group.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore.communication.comm_func import send, recv
>>> from mindspore.communication import get_rank, get_group_size
>>>
>>> np.random.seed(1)
>>> init()
>>> rank = get_rank()
>>> size = get_group_size()
>>> x = np.ones([2, 2]).astype(np.float32) * 0.01 * (rank + 1)
>>> x2 = np.ones([2, 2]).astype(np.float32)
>>>
>>>
>>> if rank < size / 2:
...     _x = ms.Tensor(x)
...     send(_x, rank + size // 2)
```

(continues on next page)

(continued from previous page)

```
... else:
...     _x2 = ms.Tensor(x2)
...     output = recv(_x2, rank - size // 2)
...     print(output)
rank1:
[[0.01  0.01]
 [0.01  0.01]]
```

11.17.13 mindspore.communication.comm_func.P2POp

class mindspore.communication.comm_func.**P2POp**(*op*, *tensor*, *peer*, *group=None*, *tag=0*, ***, *recv_dtype=None*)

Object for *batch_isend_irecv* input, to store information of `"isend"` and `"irecv"`.

Note:

- Allow pass-in recv shape rather than tensor when *op* is 'irecv'.
- *tensor* will not be modified in-place by final result.

Parameters

- **op** (*Union[str, function]*) – Only string of `"isend"` and `"irecv"` are allow. Or function of `comm_func.isend` and `comm_func.irecv` are allow.
- **tensor** (*Union[Tensor, Tuple(int)]*) – tensor for sending/receiving or receive tensor shape when *op* is `"irecv"`.
- **peer** (*int*) – remote global rank for send/receive.
- **group** (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP` , which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.
- **tag** (*int, optional*) – currently not supported yet. default: `0`.

Keyword Arguments

recv_dtype (*mindspore.dtype, optional*) – when *tensor* is a tuple shape, this arg will be used and has to be configured. default: `None`

Returns

P2POp Object.

Raises

- **ValueError** – when *op* is not string or function of 'isend' and 'irecv'.
- **TypeError** – when *tensor* is not type of Tensor or Tuple.
- **NotImplementedError** – when *tag* is not 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> send_tensor = ms.Tensor(1.)
>>> send_op = comm.comm_func.P2POp('isend', send_tensor, 1)
>>> send_op = comm.comm_func.P2POp(comm.comm_func.isend, send_tensor, 1)
>>> recv_tensor = ms.Tensor(0.)
>>> recv_op = comm.comm_func.P2POp('irecv', recv_tensor, 0)
>>> recv_op = comm.comm_func.P2POp(comm.comm_func.irecv, recv_tensor, 0)
>>> recv_op = comm.comm_func.P2POp('irecv', (), 0, recv_dtype=ms.float32)
```

11.17.14 mindspore.communication.comm_func.reduce

mindspore.communication.comm_func.**reduce**(*tensor*, *dst*, *op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*)

Reduces tensors across the processes in the specified communication group, sends the result to the target dst(global rank), and returns the tensor which is sent to the target process.

Note: Only process with destination rank receives the reduced output. Only support PyNative mode, Graph mode is not currently supported. Other processes only get a tensor with shape [1], which has no mathematical meaning.

Parameters

- **tensor** (Tensor) –The input tensor to be reduced. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **dst** (*int*) –The target rank of the process(global rank) that receives the reduced output.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like sum, prod, max, and min. On the CPU, only 'sum' is supported. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Returns

Tensor. Return the tensor in the specific rank of the process after reduction. The shape of tensor is $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> # Launch 4 processes.
>>> comm.init()
>>> dest_rank=1
>>> input_tensor = ms.Tensor(np.ones([2, 8]).astype(np.float32))
>>> output = comm.comm_func.reduce(input_tensor, dst=dest_rank)
>>> print(output)
Process with rank 1: [[4. 4. 4. 4. 4. 4. 4. 4.]
                      [4. 4. 4. 4. 4. 4. 4. 4.]],
Other proesses: [0.].
```

11.17.15 mindspore.communication.comm_func.reduce_scatter_tensor

mindspore.communication.comm_func.**reduce_scatter_tensor**(*tensor*, *op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*, *async_op=False*)

Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **tensor** (Tensor) –The input tensor to be reduced and scattered, suppose it has a shape $(N, *)$, where * means any number of additional dimensions. N must be divisible by rank_size. rank_size refers to the number of cards in the communication group.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like SUM and MAX. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.
- **async_op** (*bool, optional*) –Whether this operator should be an async operator. Default: `False`.

Returns

Tuple(Tensor, CommHandle), the output tensor has the same dtype as *input_x* with a shape of $(N/rank\_size, *)$. CommHandle is an async work handle, if *async_op* is set to True. CommHandle will be None, when *async_op* is False.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **ValueError** –If the first dimension of the input cannot be divided by the rank_size.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> comm.init()
>>> input_tensor = ms.Tensor(np.ones([8, 8]).astype(np.float32))
>>> output, _ = comm.comm_func.reduce_scatter_tensor(input_tensor)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

11.17.16 mindspore.communication.comm_func.scatter_tensor

mindspore.communication.comm_func.**scatter_tensor**(*tensor*, *src=0*, *group=GlobalComm.WORLD_COMM_GROUP*)

Scatter tensor evently across the processes in the specified communication group.

Note: The interface behavior only support Tensor input and scatter evenly, which is different from that of *pytoch.distributed.scatter*. Only the tensor in process *src* (global rank) will do scatter. Only support PyNative mode, Graph mode is not currently supported.

Parameters

- **tensor** (Tensor) –The input tensor to be scattered. The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **src** (*int, optional*) –Specifies the rank(global rank) of the process that send the tensor. And only process *src* will send the tensor.
- **group** (*str, optional*) –The communication group to work on. Default: "GlobalComm.WORLD_COMM_GROUP".

Returns

Tensor, the shape of output is $(x_1/src\_rank, x_2, ..., x_R)$. The dimension 0 of data is equal to the dimension of input tensor divided by *src*, and the other dimension keep the same.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.communication as comm
>>>
>>> # Launch 2 processes.
>>>
>>> comm.init()
>>> input = ms.Tensor(np.arange(8).reshape([4, 2]).astype(np.float32))
>>> out = comm.comm_func.scatter_tensor(tensor=input, src=0)
>>> print(out)
# rank_0
[[0. 1.]
 [2. 3.]]
# rank_1
[[4. 5.]
 [6. 7.]]
```

CHAPTER

TWELVE

MINDSPORE.DATASET

MindSpore Dataset is a high-performance data engine module specifically designed within the MindSpore framework, dedicated to providing efficient, flexible, and user-friendly data loading and preprocessing solutions for deep learning tasks. It supports multiple data formats (such as MindRecord, TFRecord, etc.) and includes a rich set of built-in public dataset interfaces, enabling users to quickly construct data pipelines. With MindSpore Dataset, users can effortlessly perform data reading, transformation, and augmentation, meeting the processing needs of various data types such as images, text, and audio.

Additionally, MindSpore Dataset offers powerful data transformation capabilities, supporting a variety of data augmentation operations (e.g., cropping, rotation, normalization, etc.), which can effectively enhance the generalization ability of models. By leveraging the efficient MindRecord data storage format, users can further optimize data reading performance, significantly accelerating large-scale data training tasks. The design of MindSpore Dataset balances flexibility and performance, supporting both single-machine and distributed training scenarios. It seamlessly integrates into the model development and training workflows of MindSpore, providing users with end-to-end efficient support from data preprocessing to model training.

- Dataset Loading(mindspore.dataset), this module provides multiple data loading methods to help users load datasets into MindSpore.
- Data Argumentation(mindspore.dataset.transforms), this module provides common data transformations in the fields of image, text and audio, and also supports customized data transformations to help users apply data argumentation online.
- MindRecord format(mindspore.mindrecord), this module provides an efficient data format that helps users to easily convert data sources to a standard format and supports high-speed reads during training.

CHAPTER

THIRTEEN

MINDSPORE.NUMPY

MindSpore Numpy package contains a set of Numpy-like interfaces, which allows developers to build models on MindSpore with similar syntax of Numpy.

MindSpore Numpy operators can be classified into four functional modules: *array generation*, *array operation*, *logic operation* and *math operation*.

Common imported modules in corresponding API examples are as follows:

```
import mindspore.numpy as np
```

Note: MindSpore numpy provides a consistent programming experience with native numpy by assembling the low-level operators. Compared with MindSpore's function and ops interfaces, it is easier for user to understand and use. However, please notice that to be more compatible with native numpy, the performance of some MindSpore numpy interfaces may be weaker than the corresponding function/ops interfaces. Users can choose which to use as needed.

13.1 Array Generation

Array generation operators are used to generate tensors.

Here is an example to generate an array:

```
import mindspore.numpy as np
import mindspore.ops as ops

input_x = np.array([1, 2, 3], np.float32)
print("input_x =", input_x)
print("type of input_x =", ops.typeof(input_x))
```

The result is as follows:

```
input_x = [1. 2. 3.]
type of input_x = Tensor[Float32]
```

Here we have more examples:

- Generate a tensor filled with the same element

  np.full can be used to generate a tensor with user-specified values:

  ```
  input_x = np.full((2, 3), 6, np.float32)
  print(input_x)
  ```

The result is as follows:

```
[[6. 6. 6.]
 [6. 6. 6.]]
```

Here is another example to generate an array with the specified shape and filled with the value of 1:

```
input_x = np.ones((2, 3), np.float32)
print(input_x)
```

The result is as follows:

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

- Generate tensors in a specified range

  Generate an arithmetic array within the specified range:

```
input_x = np.arange(0, 5, 1)
print(input_x)
```

  The result is as follows:

```
[0 1 2 3 4]
```

- Generate tensors with specific requirement

  Generate a matrix where the lower elements are 1 and the upper elements are 0 on the given diagonal:

```
input_x = np.tri(3, 3, 1)
print(input_x)
```

  The result is as follows:

```
[[1. 1. 0.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

  Another example, generate a 2-D matrix with a diagonal of 1 and other elements of 0:

```
input_x = np.eye(2, 2)
print(input_x)
```

  The result is as follows:

```
[[1. 0.]
 [0. 1.]]
```

API Name	Description	Supported Platforms
mindspore.numpy.arange	Returns evenly spaced values within a given interval.	Ascend GPU CPU
mindspore.numpy.array	Creates a tensor.	Ascend GPU CPU
mindspore.numpy.asarray	Converts the input to tensor.	Ascend GPU CPU
mindspore.numpy.asfarray	Similar to asarray, converts the input to a float tensor.	Ascend GPU CPU

continues on next page

Table 1 – continued from previous page

mindspore.numpy.bartlett	Returns the Bartlett window.	Ascend GPU CPU
mindspore.numpy.blackman	Returns the Blackman window.	Ascend GPU CPU
mindspore.numpy.copy	Returns a tensor copy of the given object.	Ascend GPU CPU
mindspore.numpy.diag	Extracts a diagonal or construct a diagonal array.	Ascend GPU CPU
mindspore.numpy.diag_indices	Returns the indices to access the main diagonal of an array.	Ascend GPU CPU
mindspore.numpy.diagflat	Creates a two-dimensional array with the flattened input as a diagonal.	Ascend GPU CPU
mindspore.numpy.diagonal	Returns specified diagonals.	Ascend GPU CPU
mindspore.numpy.empty	Returns a new array of given shape and type, without initializing entries.	Ascend GPU CPU
mindspore.numpy.empty_like	Returns a new array with the same shape and type as a given array.	Ascend GPU CPU
mindspore.numpy.eye	Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.	Ascend GPU CPU
mindspore.numpy.full	Returns a new tensor of given shape and type, filled with *fill_value*.	Ascend GPU CPU
mindspore.numpy.full_like	Returns a full array with the same shape and type as a given array.	Ascend GPU CPU
mindspore.numpy.geomspace	Returns numbers spaced evenly on a log scale (a geometric progression).	Ascend GPU CPU
mindspore.numpy.hamming	Returns the Hamming window.	Ascend GPU CPU
mindspore.numpy.hanning	Returns the Hanning window.	Ascend GPU CPU
mindspore.numpy.histogram_bin_edges	Function to calculate only the edges of the bins used by the histogram function.	Ascend GPU CPU
mindspore.numpy.identity	Returns the identity tensor.	Ascend GPU CPU
mindspore.numpy.indices	Returns an array representing the indices of a grid.	Ascend GPU CPU
mindspore.numpy.ix_	Constructs an open mesh from multiple sequences.	Ascend GPU CPU
mindspore.numpy.linspace	Returns evenly spaced values within a given interval.	Ascend GPU CPU
mindspore.numpy.logspace	Returns numbers spaced evenly on a log scale.	Ascend GPU CPU
mindspore.numpy.meshgrid	Returns coordinate matrices from coordinate vectors.	Ascend GPU CPU
mindspore.numpy.mgrid	mgrid is an *NdGrid* instance with `sparse=False`.	Ascend GPU CPU
mindspore.numpy.ogrid	ogrid is an *NdGrid* instance with `sparse=True`.	Ascend GPU CPU
mindspore.numpy.ones	Returns a new tensor of given shape and type, filled with ones.	Ascend GPU CPU
mindspore.numpy.ones_like	Returns an array of ones with the same shape and type as a given array.	Ascend GPU CPU
mindspore.numpy.pad	Pads an array.	Ascend GPU CPU
mindspore.numpy.rand	Returns a new Tensor with given shape and dtype, filled with random numbers from the uniform distribution on the interval $[0, 1)$.	Ascend GPU CPU
mindspore.numpy.randint	Return random integers from minval (inclusive) to maxval (exclusive).	Ascend GPU CPU
mindspore.numpy.randn	Returns a new Tensor with given shape and dtype, filled with a sample (or samples) from the standard normal distribution.	Ascend GPU CPU
mindspore.numpy.trace	Return the sum along diagonals of the tensor.	Ascend GPU CPU

continues on next page

Table 1 – continued from previous page

mindspore.numpy.tri	Returns a tensor with ones at and below the given diagonal and zeros elsewhere.	Ascend GPU CPU
mindspore.numpy.tril	Returns a lower triangle of a tensor.	Ascend GPU CPU
mindspore.numpy.tril_indices	Returns the indices for the lower-triangle of an (n, m) array.	Ascend GPU CPU
mindspore.numpy.tril_indices_from	Returns the indices for the lower-triangle of *arr*.	Ascend GPU CPU
mindspore.numpy.triu	Returns an upper triangle of a tensor.	Ascend GPU CPU
mindspore.numpy.triu_indices	Returns the indices for the upper-triangle of an (n, m) array.	Ascend GPU CPU
mindspore.numpy.triu_indices_from	Returns the indices for the upper-triangle of *arr*.	Ascend GPU CPU
mindspore.numpy.vander	Generates a Vandermonde matrix.	Ascend GPU CPU
mindspore.numpy.zeros	Returns a new tensor of given shape and type, filled with zeros.	Ascend GPU CPU
mindspore.numpy.zeros_like	Returns an array of zeros with the same shape and type as a given array.	Ascend GPU CPU

13.1.1 mindspore.numpy.arange

mindspore.numpy.**arange**(*start*, *stop=None*, *step=None*, *dtype=None*)

Returns evenly spaced values within a given interval.

Parameters

- **start** (*Union[int, float]*) –Start of interval. The interval includes this value. When *stop* is provided as a position argument, *start* must be given, when *stop* is a normal argument, *start* can be optional, and default: `0` . Please see additional examples below.
- **stop** (*Union[int, float], optional*) –End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of out.
- **step** (*Union[int, float], optional*) –Spacing between values. For any output *out*, this is the distance between two adjacent values, $out[i+1] - out[i]$. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype. If dtype is None, the data type of the new tensor will be inferred from start, stop and step. Default: `None` .

Returns

Tensor with evenly spaced values.

Raises

- **TypeError(PyNative Mode)** –If input arguments have types not specified above, or arguments are not given in the correct orders specified above.
- **RuntimeError(Graph Mode)** –The inputs that lead to TypeError in Pynative Mode will lead to RuntimeError in Graph Mode.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.arange(0, 5, 1))
[0 1 2 3 4]
>>> print(np.arange(3))
[0 1 2]
>>> print(np.arange(start=0, stop=3))
[0 1 2]
>>> print(np.arange(0, stop=3, step=0.5))
[0.  0.5 1.  1.5 2.  2.5]
```

13.1.2 mindspore.numpy.array

mindspore.numpy.**array**(*obj*, *dtype=None*, *copy=True*, *ndmin=0*)

Creates a tensor.

This function creates tensors from an array-like object.

Parameters

- **obj** (*Union[int, float, bool, list, tuple]*) – Input data, in any form that can be converted to a *Tensor*. This includes Tensor, list, tuple and numbers.
- **dtype** (Union[*mindspore.dtype*, str], optional) – Designated tensor dtype, can be in format of np.int32, or 'int32'. If dtype is `None`, the data type of the new tensor will be inferred from obj. Default is `None`.
- **copy** (*bool*) – If *True*, then the object is copied. Otherwise, a copy will only be made if necessary. Default: `True`.
- **ndmin** (*int*) – Specifies the minimum number of dimensions that the resulting tensor should have. Ones will be pre-pended to the shape as needed to meet this requirement. Default: `0` .

Returns

Tensor, generated tensor with the specified dtype.

Raises

- **TypeError** – If input arguments have types not specified above.
- **ValueError** – If input *obj* has different sizes at different dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.array([1,2,3]))
[1 2 3]
```

13.1.3 mindspore.numpy.asarray

mindspore.numpy.**asarray**(*a*, *dtype=None*)

Converts the input to tensor.

This function converts tensors from an array-like object.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) −Input data, in any form that can be converted to a *Tensor*. This includes Tensor, list, tuple and numbers.
- **dtype** (Union[*mindspore.dtype*, str], optional) −Designated tensor dtype, can be in format of np.int32, or 'int32'. If dtype is `None`, the data type of the new tensor will be inferred from obj. Default is `None` .

Returns

Tensor, generated tensor with the specified dtype.

Raises

- **TypeError** −If input arguments have types not specified above.
- **ValueError** −If input *a* has different sizes at different dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.asarray([1,2,3]))
[1 2 3]
```

13.1.4 mindspore.numpy.asfarray

mindspore.numpy.**asfarray**(*a*, *dtype=mstype.float32*)

Similar to asarray, converts the input to a float tensor.

If non-float dtype is defined, this function will return a float32 tensor instead.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) −Input data, in any form that can be converted to a *Tensor*. This includes Tensor, list, tuple and numbers.
- **dtype** (Union[*mindspore.dtype*, str], optional) −Designated tensor dtype, can be in format of np.int32, or 'int32'. If dtype is `None`, the data type of the new tensor will be inferred from *a*. Default is `mstype.float32`.

Returns

Tensor, generated tensor with the specified float dtype.

Raises

- **TypeError** −If input arguments have types not specified above.
- **ValueError** −If input *a* has different sizes at different dimensions.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.asfarray([1,2,3]))
[1. 2. 3.]
```

13.1.5 mindspore.numpy.bartlett

mindspore.numpy.**bartlett**(*M*)

Returns the Bartlett window. The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

Parameters

M (*int*) −Number of points in the output window. If zero or less, an empty array is returned.

Returns

Tensor, the triangular window, with the maximum value normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

Raises

TypeError −If *M* is not an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.bartlett(12))
[0.         0.18181819 0.36363637 0.5454545  0.72727275 0.9090909
0.9090909  0.72727275 0.5454545  0.36363637 0.18181819 0.        ]
```

13.1.6 mindspore.numpy.blackman

mindspore.numpy.**blackman**(*M*)

Returns the Blackman window. The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

Parameters

M (*int*) −Number of points in the output window. If zero or less, an empty array is returned.

Returns

Tensor, the window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

Raises

TypeError −If *M* is not an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.blackman(12))
[-1.4901161e-08  3.2606430e-02  1.5990365e-01  4.1439798e-01
7.3604518e-01  9.6704674e-01  9.6704674e-01  7.3604518e-01
4.1439798e-01  1.5990365e-01  3.2606430e-02 -1.4901161e-08]
```

13.1.7 mindspore.numpy.copy

mindspore.numpy.**copy**(*a*)

Returns a tensor copy of the given object.

Parameters

a (*Union[int, float, bool, list, tuple, Tensor]*) – Input data, in any form that can be converted to a Tensor. This includes Tensor, list, tuple and numbers.

Returns

Tensor, has the same data as *a*.

Raises

- **TypeError** – If input *a* has type not specified above.
- **ValueError** – If input *a* has different sizes at different dimensions.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((2,2))
>>> print(np.copy(x))
[[1. 1.]
 [1. 1.]]
```

13.1.8 mindspore.numpy.diag

mindspore.numpy.**diag**(*v*, *k=0*)

Extracts a diagonal or construct a diagonal array.

Parameters

- **v** (Tensor) – If *v* is a 2-D array, return a copy of its *k-th* diagonal. If *v* is a 1-D array, return a 2-D array with v on the *k-th* diagonal.
- **k** (*int, optional*) – Diagonal in question. The default is 0. Use `k>0` for diagonals above the main diagonal, and `k<0` for diagonals below the main diagonal.

Returns

Tensor, the extracted diagonal or constructed diagonal array.

Raises

ValueError – If input is not 1-D or 2-D.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(9).reshape((3,3))
>>> print(x)
[[0 1 2]
[3 4 5]
[6 7 8]]
>>> output = np.diag(x)
>>> print(output)
[0 4 8]
>>> output = np.diag(x, k=1)
>>> print(output)
[1 5]
>>> output = np.diag(x, k=-1)
>>> print(output)
[3 7]
```

13.1.9 mindspore.numpy.diag_indices

mindspore.numpy.**diag_indices**(*n*, *ndim=2*)

Returns the indices to access the main diagonal of an array.

This returns a tuple of indices that can be used to access the main diagonal of an array a with `a.ndim >= 2` dimensions and shape *(n, n, ···, n)*. For `a.ndim = 2` this is the usual diagonal, for `a.ndim > 2` this is the set of indices to access `a[i, i, ..., i]` for `i = [0..n-1]`.

Parameters

- **n** (*int*) – The size, along each dimension, of the arrays for which the returned indices can be used.
- **ndim** (*int, optional*) – The number of dimensions.

Returns

Tuple of Tensor.

Raises

TypeError – If input are not integers.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.diag_indices(5, 3)
>>> print(output)
(Tensor(shape=[5], dtype=Int32, value= [0, 1, 2, 3, 4]),
Tensor(shape=[5], dtype=Int32, value= [0, 1, 2, 3, 4]),
Tensor(shape=[5], dtype=Int32, value= [0, 1, 2, 3, 4]))
```

13.1.10 mindspore.numpy.diagflat

mindspore.numpy.**diagflat**(*v*, *k=0*)

Creates a two-dimensional array with the flattened input as a diagonal.

Note: On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **v** (Tensor) −Input data, which is flattened and set as the *k-th* diagonal of the output.
- **k** (*int, optional*) −Diagonal to set; 0, the default, corresponds to the "main" diagonal, a positive (negative) *k* giving the number of the diagonal above (below) the main.

Returns

Tensor, The 2-D output array.

Raises

TypeError −If the input is not a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.diagflat(np.asarray([[1,2], [3,4]]))
>>> print(output)
[[1 0 0 0]
[0 2 0 0]
[0 0 3 0]
[0 0 0 4]]
>>> output = np.diagflat(np.asarray([1,2]), 1)
>>> print(output)
[[0 1 0]
[0 0 2]
[0 0 0]]
```

13.1.11 mindspore.numpy.diagonal

mindspore.numpy.**diagonal**(*a*, *offset=0*, *axis1=0*, *axis2=1*)

Returns specified diagonals.

If *a* is 2-D, returns the diagonal of *a* with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

Parameters

- **a** (Tensor) – Array from which the diagonals are taken.
- **offset** (*int, optional*) – Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal.
- **axis1** (*int, optional*) – Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).
- **axis2** (*int, optional*) – Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis.

Returns

Tensor, if *a* is 2-D, then *a* 1-D array containing the diagonal. If `a.ndim > 2`, then the dimensions specified by *axis1* and *axis2* are removed, and a new axis inserted at the end corresponding to the diagonal.

Raises

ValueError – If the input tensor has less than two dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(4).reshape(2,2).astype(np.float32)
>>> print(a)
[[0. 1.]
[2. 3.]]
>>> output = np.diagonal(a)
>>> print(output)
[0. 3.]
>>> output = np.diagonal(a, 1)
>>> print(output)
[1.]
>>> a = np.arange(8).reshape(2, 2, 2).astype(np.float32)
>>> print(a)
[[[0. 1.]
[2. 3.]]
[[4. 5.]
[6. 7.]]]
>>> output = np.diagonal(a, 0, 0, 1)
>>> print(output)
[[0. 6.]
[1. 7.]]
```

13.1.12 mindspore.numpy.empty

mindspore.numpy.**empty**(*shape*, *dtype=mstype.float32*)

Returns a new array of given shape and type, without initializing entries.

Note: Numpy argument *order* is not supported. Object arrays are not supported.

Parameters

- **shape** (*Union[int, tuple(int)]*) –Shape of the empty array, e.g., (2, 3) or 2.
- **dtype** (*mindspore.dtype*, optional) –Desired output data-type for the array, e.g, mstype.int8. Default: `mstype.float32`.

Returns

Tensor, array of uninitialized (arbitrary) data of the given shape and dtype.

Raises

TypeError –If the input shape or dtype is invalid.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.empty((2, 3))
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]]
```

13.1.13 mindspore.numpy.empty_like

mindspore.numpy.**empty_like**(*prototype*, *dtype=None*, *shape=None*)

Returns a new array with the same shape and type as a given array.

Note: Input array must have the same size across a dimension. If *prototype* is not a Tensor, dtype is float32 by default if not provided.

Parameters

- **prototype** (*Union[Tensor, list, tuple]*) –The shape and data-type of *prototype* define these same attributes of the returned array.
- **dtype** (*mindspore.dtype*, optional) –Overrides the data type of the result.
- **shape** (*int or sequence of ints, optional*) –Overrides the shape of the result.

Returns

Tensor, array of uninitialized (arbitrary) data with the same shape and type as *prototype*.

Raises

ValueError – If *prototype* is not a Tensor, list or tuple.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((4,1,2))
>>> output = np.empty_like(a)
>>> print(output)
[[[0. 0.]]
 [[0. 0.]]
 [[0. 0.]]
 [[0. 0.]]]
```

13.1.14 mindspore.numpy.eye

`mindspore.numpy.`**`eye`**(*N*, *M=None*, *k=0*, *dtype=mstype.float32*)

Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.

Parameters

- **N** (*int*) – Number of rows in the output, must be larger than 0.
- **M** (*int, optional*) – Number of columns in the output. If is `None`, default: `N` , if defined, must be larger than 0. Default: `None`.
- **k** (*int, optional*) – Index of the diagonal: `0` (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal. Default: `0` .
- **dtype** (Union[*mindspore.dtype*, str], optional) – Designated tensor dtype. Default: `mstype.float32` .

Returns

A tensor of shape (N, M). A tensor where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

Raises

TypeError – If input arguments have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.eye(2, 2))
[[1. 0.]
[0. 1.]]
```

13.1.15 mindspore.numpy.full

mindspore.numpy.**full**(*shape*, *fill_value*, *dtype=None*)

Returns a new tensor of given shape and type, filled with *fill_value*.

Parameters

- **shape** (*Union[int, tuple(int), list(int)]*) –Shape of the new tensor, e.g., $(2, 3)$ or 2.
- **fill_value** (*Union[int, float, bool, list, tuple]*) –Scalar or array_like fill value.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype, if *dtype* is `None`, the data type of the new tensor will be inferred from *fill_value*. Default is `None`.

Returns

Tensor, with the designated shape and dtype, filled with *fill_value*.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If *shape* has entries < 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.full((2,2), True))
[[True True]
[True True]]
```

13.1.16 mindspore.numpy.full_like

mindspore.numpy.**full_like**(*a*, *fill_value*, *dtype=None*, *shape=None*)

Returns a full array with the same shape and type as a given array.

Note:

- Input array must have the same size across a dimension.
- If *a* is not a Tensor, dtype is float32 by default if not provided.

Parameters

- **a** (*Union[Tensor, list, tuple]*) –The shape and data-type of *a* define these same attributes of the returned array.
- **fill_value** (*scalar*) –Fill value.
- **dtype** (*mindspore.dtype*, optional) –Overrides the data type of the result.
- **shape** (*int, sequence of ints, optional*) –Overrides the shape of the result.

Returns

Tensor, array of fill_value with the same shape and type as *a*.

Raises

ValueError –If *a* is not a Tensor, list or tuple.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((4,1,2))
>>> output = np.full_like(a, 0.5)
>>> print(output)
[[[0.5 0.5]]
[[0.5 0.5]]
[[0.5 0.5]]
[[0.5 0.5]]]
```

13.1.17 mindspore.numpy.geomspace

`mindspore.numpy.`**geomspace**(*start*, *stop*, *num=50*, *endpoint=True*, *dtype=None*, *axis=0*)

Returns numbers spaced evenly on a log scale (a geometric progression).

This is similar to logspace, but with endpoints specified directly. Each output sample is a constant multiple of the previous.

Parameters

- **start** (*Union[int, list(int), tuple(int), tensor]*) –The starting value of the sequence.
- **stop** (*Union[int, list(int), tuple(int), tensor]*) –The final value of the sequence, unless endpoint is False. In that case, num + 1 values are spaced over the interval in log-space, of which all but the last (a sequence of length num) are returned.
- **num** (*int, optional*) –Number of samples to generate. Default: `50` .
- **endpoint** (*bool, optional*) –If True, *stop* is the last sample. Otherwise, it is not included. Default: `True` .
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype, can be in format of np.float32, or *float32*.If *dtype* is None, infer the data type from other input arguments. Default: `None` .
- **axis** (*int, optional*) –The axis in the result to store the samples. Relevant only if start or stop is array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end. Default: `0` .

Returns

Tensor, with samples equally spaced on a log scale.

Raises

TypeError – If input arguments have types not specified above.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.geomspace(1, 256, num=9)
>>> print(output)
[  1.   2.   4.   8.  16.  32.  64. 128. 256.]
>>> output = np.geomspace(1, 256, num=8, endpoint=False)
>>> print(output)
[  1.   2.   4.   8.  16.  32.  64. 128.]
```

13.1.18 mindspore.numpy.hamming

mindspore.numpy.**hamming**(*M*)

Returns the Hamming window. The Hamming window is a taper formed by using a weighted cosine.

Parameters

M (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

Returns

Tensor, the window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

Raises

TypeError – If *M* is not an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.hamming(12))
[0.08000001 0.15302339 0.34890914 0.6054648  0.841236   0.9813669
0.9813668  0.8412359  0.6054647  0.34890908 0.15302327 0.08000001]
```

13.1.19 mindspore.numpy.hanning

mindspore.numpy.**hanning**(*M*)

Returns the Hanning window. The Hanning window is a taper formed by using a weighted cosine.

Parameters

M (*int*) – Number of points in the output window. If zero or less, an empty array is returned.

Returns

Tensor, the window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

Raises

TypeError – If *M* is not an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.hanning(12))
[0.         0.07937324 0.29229254 0.5711574  0.8274304  0.9797465
0.97974646 0.82743025 0.5711573  0.29229245 0.07937312 0.        ]
```

13.1.20 mindspore.numpy.histogram_bin_edges

mindspore.numpy.**histogram_bin_edges**(*a*, *bins=10*, *range=None*, *weights=None*)

Function to calculate only the edges of the bins used by the histogram function.

Note: String values for *bins* is not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input data. The histogram is computed over the flattened array.
- **bins** (*Union[int, tuple, list, Tensor], optional*) – If *bins* is an int, it defines the number of equal-width bins in the given range (10, by default). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.
- **range** (*(float, float), optional*) – The lower and upper range of the bins. If not provided, *range* is simply `(a.min(), a.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. Default: `None` .
- **weights** (*Union[int, float, bool, list, tuple, Tensor], optional*) – An array of weights, of the same shape as *a*. Each value in *a* only contributes its associated weight towards the bin count (instead of 1). This is currently not used by any of the bin estimators, but may be in the future. Default: `None` .

Returns

Tensor, the edges to pass into *histogram*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

TypeError – If *bins* is an array and not one-dimensional.

Examples

```
>>> import mindspore.numpy as np
>>> arr = np.array([0, 0, 0, 1, 2, 3, 3, 4, 5])
>>> print(np.histogram_bin_edges(arr, bins=2))
[0.  2.5 5. ]
```

13.1.21 mindspore.numpy.identity

mindspore.numpy.**identity**(*n*, *dtype=mstype.float32*)

Returns the identity tensor.

Parameters

- **n** (*int*) – Number of rows and columns in the output, must be larger than 0.
- **dtype** (Union[*mindspore.dtype*, str], optional) – Designated tensor dtype, default is *mstype.float32*.

Returns

A tensor of shape *(n, n)*, where all elements are equal to zero, except for the diagonal, whose values are equal to one.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

TypeError – If input arguments have types not specified above.

Examples

```
>>> import mindspore.numpy as np
>>> print(np.identity(2))
[[1. 0.]
[0. 1.]]
```

13.1.22 mindspore.numpy.indices

mindspore.numpy.**indices**(*dimensions*, *dtype=mstype.int32*, *sparse=False*)

Returns an array representing the indices of a grid.

Computes an array where the subarrays contain index values 0, 1, ···varying only along the corresponding axis.

Parameters

- **dimensions** (*Union[list(int), tuple]*) –The shape of the grid.
- **dtype** (*mindspore.dtype*, optional) –Data type of the result. Default: `mstype.int32`.
- **sparse** (*boolean, optional*) –Default: `False` . Return a sparse representation of the grid instead of a dense representation.

Returns

Tensor or tuple of Tensor, If *sparse* is False, returns one array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`. If sparse is True, returns a tuple of arrays, with `grid[i].shape = (1, ..., 1, dimensions[i], 1, ..., 1)` with `dimensions[i]` in the *ith* place

Raises

TypeError –If input dimensions is not a tuple or list.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> grid = np.indices((2, 3))
>>> print(grid)
[Tensor(shape=[2, 3], dtype=Int32, value=
[[0, 0, 0],
[1, 1, 1]]), Tensor(shape=[2, 3], dtype=Int32, value=
[[0, 1, 2],
[0, 1, 2]])]
```

13.1.23 mindspore.numpy.ix

mindspore.numpy.**ix_**(**args*)

Constructs an open mesh from multiple sequences.

This function takes *N* 1-D sequences and returns *N* outputs with *N* dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions. Using ix_ one can quickly construct index arrays that will index the cross product. `a[np.ix_([1,3],[2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]`.

Note: Boolean masks are not supported.

Parameters

***args** (`Tensor`) –1-D sequences.

Returns

Tuple of Tensor, *N* arrays with *N* dimensions each, with *N* the number of input sequences. Together these arrays form an open mesh.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore.numpy as np
>>> ixgrid = np.ix_(np.array([0, 1]), np.array([2, 4]))
>>> print(ixgrid)
(Tensor(shape=[2, 1], dtype=Int32, value=
[[0],
[1]]), Tensor(shape=[1, 2], dtype=Int32, value=
[[2, 4]]))
```

13.1.24 mindspore.numpy.linspace

`mindspore.numpy.`**linspace**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *axis=0*)

Returns evenly spaced values within a given interval.

Parameters

- **start** (*Union[int, list(int), tuple(int), tensor]*) –The starting value of the sequence.
- **stop** (*Union[int, list(int), tuple(int), tensor]*) –The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of *num + 1* evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.
- **num** (*int, optional*) –Number of samples to generate. Default: `50` .
- **endpoint** (*bool, optional*) –If True, *stop* is the last sample. Otherwise, it is not included. Default: `True` .
- **retstep** (*bool, optional*) –If True, return (*samples*, *step*), where *step* is the spacing between samples.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype, If *dtype* is None, infer the data type from other input arguments. Default: `None` .
- **axis** (*int, optional*) –The axis in the result to store the samples. Relevant only if start or stop are array-like. By default, the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end. Default: `0` .

Returns

Tensor, with *num* equally spaced samples in the closed interval $[start, stop]$ or the half-open interval $[start, stop)$ (depending on whether *endpoint* is True or False).

Step, the size of spacing between samples, only returned if *retstep* is True.

Raises

TypeError –If input arguments have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.linspace(0, 5, 6))
[0. 1. 2. 3. 4. 5.]
```

13.1.25 mindspore.numpy.logspace

`mindspore.numpy.`**`logspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *base=10.0*, *dtype=None*, *axis=0*)

Returns numbers spaced evenly on a log scale.

In linear space, the sequence starts at base ** start (base to the power of start) and ends with base ** stop (see endpoint below).

Parameters

- **start** (*Union[int, list(int), tuple(int), tensor]*) –`base ** start` is the starting value of the sequence.
- **stop** (*Union[int, list(int), tuple(int), tensor]*) –`base ** stop` is the final value of the sequence, unless *endpoint* is False. In that case, `num + 1` values are spaced over the interval in log-space, of which all but the last (a sequence of length num) are returned.
- **num** (*int, optional*) –Number of samples to generate. Default: `50` .
- **endpoint** (*bool, optional*) –If True, *stop* is the last sample. Otherwise, it is not included. Default: `True` .
- **base** (*Union[int, float], optional*) –The base of the log space. The step size between the elements in $ln(samples)/ln(base)$ (or $log_{base}(samples)$) is uniform. Default: `10.0` .
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype. If *dtype* is None, infer the data type from other input arguments. Default: `None` .
- **axis** (*int, optional*) –The axis in the result to store the samples. Relevant only if start or stop is array-like. By default, the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end. Default: `0` .

Returns

Tensor, equally spaced on a log scale.

Raises

TypeError –If input arguments have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.logspace(0, 5, 6, base=2.0))
[ 1.  2.  4.  8. 16. 32.]
```

13.1.26 mindspore.numpy.meshgrid

mindspore.numpy.**meshgrid**(**xi*, *sparse=False*, *indexing='xy'*)

Returns coordinate matrices from coordinate vectors.

Make *N-D* coordinate arrays for vectorized evaluations of *N-D* scalar/vector fields over *N-D* grids, given one-dimensional coordinate arrays *x1, x2,···, xn*.

Note: Numpy argument copy is not supported, and a copy is always returned.

Parameters

- ***xi** (Tensor) – 1-D arrays representing the coordinates of a grid.
- **indexing** (*'xy', 'ij', optional*) – Cartesian ('xy', default) or matrix ('ij') indexing of output. In the 2-D case with inputs of length *M* and *N*, the outputs are of shape *(N, M)* for 'xy' indexing and *(M, N)* for 'ij' indexing. In the 3-D case with inputs of length *M*, *N* and *P*, outputs are of shape *(N, M, P)* for 'xy' indexing and *(M, N, P)* for 'ij' indexing.
- **sparse** (*bool, optional*) – If True a sparse grid is returned in order to conserve memory. Default is False.

Returns

Tuple of tensors, for vectors *x1, x2,···, xn* with lengths `Ni=len(xi)`, return *(N1, N2, N3,···Nn)* shaped arrays if `indexing='ij'` or *(N2, N1, N3,···Nn)* shaped arrays if `indexing='xy'` with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

Raises

TypeError – If the input is not a tensor, or sparse is not boolean, or indexing is not 'xy' or 'ij'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.linspace(0, 1, 3)
>>> y = np.linspace(0, 1, 2)
>>> xv, yv = np.meshgrid(x, y)
>>> print(xv)
[[0.  0.5 1. ]
[0.  0.5 1. ]]
>>> print(yv)
[[0.  0.  0.]
[1.  1.  1.]]
>>> xv, yv = np.meshgrid(x, y, sparse=True)
```

(continues on next page)

(continued from previous page)

```
>>> print(xv)
[[0.  0.5  1. ]]
>>> print(yv)
[[0.]
[1.]]
```

13.1.27 mindspore.numpy.mgrid

mindspore.numpy.**mgrid**()

mgrid is an *NdGrid* instance with `sparse=False`.

The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive. However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

Note: Not supported in graph mode. Unlike Numpy, if the step length is a complex number with a real component, the step length is handled as equivalent to `int(abs(step))`.

Returns

Tensor or tuple of tensor, a meshgrid.

Raises

TypeError – If slicing indices are not integers.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.mgrid[0:5, 0:5]
>>> print(output)
[[[0 0 0 0 0]
[1 1 1 1 1]
[2 2 2 2 2]
[3 3 3 3 3]
[4 4 4 4 4]]
[[0 1 2 3 4]
[0 1 2 3 4]
[0 1 2 3 4]
[0 1 2 3 4]
[0 1 2 3 4]]]
>>> output = mgrid[-1:1:5j]
>>> print(output)
[-1.  -0.5  0.   0.5  1. ]
```

13.1.28 mindspore.numpy.ogrid

mindspore.numpy.**ogrid**()

ogrid is an *NdGrid* instance with `sparse=True`.

The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive. However, if the step length is a complex number (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value is inclusive.

Note: Not supported in graph mode. Unlike Numpy, if the step length is a complex number with a real component, the step length is handled as equivalent to `int(abs(step))`.

Raises

TypeError –If slicing indices are not integers.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.ogrid[0:5,0:5]
>>> print(output)
[Tensor(shape=[5, 1], dtype=Int32, value=
[[0],
[1],
[2],
[3],
[4]]), Tensor(shape=[1, 5], dtype=Int32, value=
[[0, 1, 2, 3, 4]])]
>>> output = ogrid[-1:1:5j]
>>> print(output)
[-1.  -0.5  0.   0.5  1. ]
```

13.1.29 mindspore.numpy.ones

mindspore.numpy.**ones**(*shape*, *dtype=mstype.float32*)

Returns a new tensor of given shape and type, filled with ones.

Parameters

- **shape** (*Union[int, tuple, list]*) –the shape of the new tensor.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype. Default is `mstype.float32`.

Returns

Tensor, with the designated *shape* and *dtype*, filled with ones.

Raises

- **TypeError** –If input arguments have types not specified above.

- **ValueError** –If *shape* entries have values < 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.ones((2,2)))
[[1. 1.]
[1. 1.]]
```

13.1.30 mindspore.numpy.ones_like

mindspore.numpy.**ones_like**(*a*, *dtype=None*, *shape=None*)

Returns an array of ones with the same shape and type as a given array.

Note: Input array must have the same size across a dimension. If *a* is not a Tensor, dtype is float32 by default if not provided.

Parameters

- **a** (*Union[Tensor, list, tuple]*) –The shape and data-type of a define these same attributes of the returned array.
- **dtype** (*mindspore.dtype*, optional) –Overrides the data type of the result. Default: None.
- **shape** (*int, sequence of ints, optional*) –Overrides the shape of the result. Default: None.

Returns

Tensor, array of ones with the same shape and type as *a*.

Raises

ValueError –If *a* is not a Tensor, list or tuple.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((4,1,2))
>>> output = np.ones_like(a)
>>> print(output)
[[[1. 1.]]
[[1. 1.]]
[[1. 1.]]
[[1. 1.]]]
```

13.1.31 mindspore.numpy.pad

mindspore.numpy.**pad**(*arr*, *pad_width*, *mode='constant'*, *stat_length=None*, *constant_values=0*, *end_values=0*, *reflect_type='even'*, ***kwargs*)

Pads an array.

Note: Currently, *median* mode is not supported. *reflect* and *symmetric* mode only supports GPU backend.

Parameters

- **arr** (*Union[list, tuple, Tensor]*) –The array to pad.
- **pad_width** (*Union[int, tuple, list]*) –Number of values padded to the edges of each axis. `((before_1, after_1), ... (before_N, after_N))` creates unique pad widths for each axis. `((before, after),)` yields same before and after pad for each axis. `(pad,)` or int is a shortcut for `before = after = pad width` for all axes.
- **mode** (*string, optional*) –One of the following string values:
  - constant (default): Pads with a constant value.
  - edge: Pads with the edge values of *arr*.
  - linear_ramp: Pads with the linear ramp between end_value and the *arr* edge value.
  - maximum: Pads with the maximum value of all or part of the vector along each axis.
  - mean: Pads with the mean value of all or part of the vector along each axis.
  - median: Pads with the median value of all or part of the vector along each axis.
  - minimum: Pads with the minimum value of all or part of the vector along each axis.
  - reflect: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
  - symmetric: Pads with the reflection of the vector mirrored along the edge of the *arr*.
  - wrap: Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.
  - empty: Pads with undefined values.
  - <function>: The padding function, if used, should modify and return a new 1-d tensor. It has the following signature: `padding_func(tensor, iaxis_pad_width, iaxis, kwargs)`
- **stat_length** (*Union[tuple, list, int], optional*) –Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value. `((before_1, after_1), ... (before_N, after_N))` creates unique statistic lengths for each axis. `((before, after),)` yields same before and after statistic lengths for each axis. `(stat_length,)` or int is a shortcut for `before = after = statistic length` for all axes. Default: `None`, to use the entire axis.
- **constant_values** (*Union[tuple, list, int], optional*) –Used in `constant mode`. The values to set the padded values for each axis. `((before_1, after_1), ... (before_N, after_N))` creates unique pad constants for each axis. `((before, after),)` yields same before and after constants for each axis. `(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes. Default: `0` .
- **end_values** (*Union[tuple, list, int], optional*) –Used in 'linear_ramp'. The values used for the ending value of the linear_ramp and that will form the edge of the padded *arr*. `((before_1, after_1), ... (before_N, after_N))` unique end values for each axis. `((before, after),)` yields same before and after end values for each axis. `(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes. Default: `0` .

- **reflect_type** (*string, optional*) –'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extended part of the *arr* is created by subtracting the reflected values from two times the edge value.
- **kwargs** (*anytype, optional*) –Any keyword arguments that will be used only in <function> mode.

Returns

Padded tensor of rank equal to *arr* with shape increased according to *pad_width*.

Raises

- **TypeError** –If *arr*, *pad_width*, *stat_length*, *constant_values* or *end_values* have types not specified above.
- **ValueError** –If *mode* cannot be recognized, or if *pad_width*, *stat_length*, *constant_values*, *end_values* cannot broadcast to `(arr.ndim, 2)`, or if keyword arguments got unexpected inputs.
- **NotImplementedError** –If mode is function or 'median'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> tensor = np.array([1., 2., 3., 4., 5.])
>>> print(np.pad(tensor, (3, 4)))
[0. 0. 0. 1. 2. 3. 4. 5. 0. 0. 0. 0.]
>>> print(np.pad(tensor, (3, 4), mode="wrap"))
[3. 4. 5. 1. 2. 3. 4. 5. 1. 2. 3. 4.]
>>> print(np.pad(tensor, (3, 4), mode="linear_ramp", end_values=(10, 10)))
[10.    7.    4.    1.    2.    3.    4.    5.    6.25  7.5   8.75 10.  ]
```

13.1.32 mindspore.numpy.rand

`mindspore.numpy.`**rand**(**shape*, *dtype=mstype.float32*)

Returns a new Tensor with given shape and dtype, filled with random numbers from the uniform distribution on the interval $[0, 1)$.

Parameters

- ***shape** (*Union[int, tuple(int), list(int)]*) –Shape of the new tensor, e.g., $(2, 3)$ or 2.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype, it must be float type. Default is `mindspore.float32`.

Returns

Tensor, with the designated shape and dtype, filled with random numbers from the uniform distribution on the interval $[0, 1)$.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If *dtype* is not float type.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> from mindspore import set_seed
>>> set_seed(1)
>>> print(np.rand((2,3)))
[[4.1702199e-01 9.9718481e-01 7.2032452e-01]
[9.3255734e-01 1.1438108e-04 1.2812445e-01]]
```

13.1.33 mindspore.numpy.randint

mindspore.numpy.**randint**(*minval*, *maxval=None*, *shape=None*, *dtype=mstype.int32*)

Return random integers from minval (inclusive) to maxval (exclusive). Return random integers from the discrete uniform distribution of the specified dtype in the "half-open" interval $[minval, maxval)$. If maxval is None (the default), the value range will be $[0, minval)$, in this case, minval must be greater than 0.

Parameters

- **minval** (*Union[int]*) –Start value of interval. The interval includes this value. When *maxval* is None, *minval* must be greater than 0. When *maxval* is not None, *minval* must be less than *maxval*.
- **maxval** (*Union[int], optional*) –End value of interval. The interval does not include this value.
- **shape** (*Union[int, tuple(int)]*) –Shape of the new tensor, e.g., $(2, 3)$ or 2.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype, it must be int type. Default is *mindspore.int32*.

Returns

Tensor, with the designated shape and dtype, filled with random integers from minval (inclusive) to maxval (exclusive).

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If input arguments have values not specified above.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> from mindspore import set_seed
>>> set_seed(1)
>>> print(np.randint(1, 10, (2,3)))
[[4 9 7]
[9 1 2]]
```

13.1.34 mindspore.numpy.randn

mindspore.numpy.**randn**(**shape*, *dtype=mstype.float32*)

Returns a new Tensor with given shape and dtype, filled with a sample (or samples) from the standard normal distribution.

Parameters

- ***shape** (*Union[int, tuple(int), list(int)]*) – Shape of the new tensor, e.g., (2, 3) or 2.
- **dtype** (Union[*mindspore.dtype*, str], optional) – Designated tensor dtype, it must be float type. Default is `mindspore.float32`.

Returns

Tensor, with the designated shape and dtype, filled with a sample (or samples) from the "standard normal" distribution.

Raises

- **TypeError** – If input arguments have types not specified above.
- **ValueError** – If *dtype* is not float type.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> from mindspore import set_seed
>>> set_seed(1)
>>> print(np.randn((2,3)))
[[ 0.30639967 -0.42438635 -0.20454668]
[-0.4287376   1.3054721   0.64747655]]
```

13.1.35 mindspore.numpy.trace

mindspore.numpy.**trace**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*)

Return the sum along diagonals of the tensor.

Note:

- *trace* is currently only used in *mindscience* scientific computing scenarios and dose not support other usage scenarios.
- *trace* is not supported on Windows platform yet.

Parameters

- **a** (`Tensor`) – A matrix to be calculated.
- **offset** (*int, optional*) – Offset of the diagonal from the main diagonal. Can be positive or negative. Default: `0`.
- **axis1** (*int, optional*) – Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Default: `0`.
- **axis2** (*int, optional*) – Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Default: `1`.

- **dtype** (*mindspore.dtype*, optional) –Overrides the dtype of the output Tensor if not `None`. Default: `None`.

Returns

Tensor, the sum along diagonals. If *a* is 2-D, the sum along the diagonal is returned. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of *a* with *axis1* and *axis2* removed.

Raises

- **ValueError** –If *a* has less than two dimensions.
- **ValueError** –If *axis1* or *axis2* is not in [-dims, dims), which dims is dimension of *a*.
- **ValueError** –If axes specified by *axis1* and *axis2* are same.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.numpy import trace
>>> x = Tensor(np.eye(3, dtype=np.float32))
>>> print(trace(x))
3.0
```

13.1.36 mindspore.numpy.tri

`mindspore.numpy.tri`(*N*, *M=None*, *k=0*, *dtype=mstype.float32*)

Returns a tensor with ones at and below the given diagonal and zeros elsewhere.

Parameters

- **N** (*int*) –Number of rows in the array.
- **M** (*int, optional*) –Number of columns in the array. By default, *M* is taken equal to N. Default: `None` .
- **k** (*int, optional*) –The sub-diagonal at and below which the array is filled. $k = 0$ is the main diagonal, while $k < 0$ is below it, and $k > 0$ is above. Default: `0` .
- **dtype** (*mindspore.dtype*, optional) –Data type of the returned array. Default: `mstype.float32` .

Returns

Tensor with shape *(N, M)*, with its lower triangle filled with ones and zeros elsewhere; in other words $T[i, j] = 1$ for $j <= i + k$, 0 otherwise.

Raises

TypeError –If input arguments have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.tri(3, 3, 1)
>>> print(output)
[[1. 1. 0.]
[1. 1. 1.]
[1. 1. 1.]]
```

13.1.37 mindspore.numpy.tril

mindspore.numpy.**tril**(*m*, *k=0*)

Returns a lower triangle of a tensor.

Returns a copy of a tensor with elements above the *k-th* diagonal zeroed.

Parameters

- **m** (*Union[Tensor, list, tuple]*) – The shape and data-type of *m* define these same attributes of the returned tensor.
- **k** (*int, optional*) – Diagonal above which to zero elements. $k = 0$ (the default) is the main diagonal, $k < 0$ is below it and $k > 0$ is above.

Returns

Lower triangle of *m*, of same shape and data-type as *m*.

Supported Platforms:

Ascend GPU CPU

Raises

- **TypeError** – If input arguments have types not specified above.
- **ValueError** – If input *m*'s rank < 1.

Examples

```
>>> import mindspore.numpy as np
>>> output = np.tril(np.ones((3, 3)))
>>> print(output)
[[1. 0. 0.]
[1. 1. 0.]
[1. 1. 1.]]
```

13.1.38 mindspore.numpy.tril_indices

mindspore.numpy.**tril_indices**(*n*, *k=0*, *m=None*)

Returns the indices for the lower-triangle of an (n, m) array.

Parameters

- **n** (*int*) –The size of the arrays for which the returned indices will be valid.
- **k** (*int, optional*) –Diagonal offset, default: `0` .
- **m** (*int, optional*) –The column dimension of the arrays for which the returned arrays will be valid. By default *m* is taken equal to *n*. Default: `None` .

Returns

The indices for the triangle. The returned tuple contains two tensors, each with the indices along one dimension of the tensor.

Raises

TypeError –If *n*, *k*, *m* are not numbers.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.tril_indices(3))
(Tensor(shape=[6], dtype=Int32, value= [0, 1, 1, 2, 2, 2]),
Tensor(shape=[6], dtype=Int32, value= [0, 0, 1, 0, 1, 2]))
```

13.1.39 mindspore.numpy.tril_indices_from

mindspore.numpy.**tril_indices_from**(*arr*, *k=0*)

Returns the indices for the lower-triangle of *arr*.

Parameters

- **arr** (*Union[Tensor, list, tuple]*) –2-dimensional array.
- **k** (*int, optional*) –Diagonal offset, default: `0` .

Returns

triu_indices_from, tuple of 2 tensor, shape(N) Indices for the lower-triangle of *arr*.

Raises

- **TypeError** –If *arr* cannot be converted to tensor, or *k* is not a number.
- **ValueError** –If *arr* cannot be converted to a 2-dimensional tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> tensor = np.ones((3,3))
>>> print(np.tril_indices_from(tensor))
(Tensor(shape=[6], dtype=Int32, value= [0, 1, 1, 2, 2, 2]),
 Tensor(shape=[6], dtype=Int32, value= [0, 0, 1, 0, 1, 2]))
```

13.1.40 mindspore.numpy.triu

mindspore.numpy.**triu**(*m*, *k=0*)

Returns an upper triangle of a tensor.

Returns a copy of a tensor with elements below the *k-th* diagonal zeroed.

Parameters

- **m** (*Union[Tensor, list, tuple]*) –The shape and data-type of *m* define these same attributes of the returned tensor.
- **k** (*int, optional*) –Diagonal below which to zero elements. $k = 0$ (the default) is the main diagonal, $k < 0$ is below it and $k > 0$ is above.

Returns

Upper triangle of *m*, of same shape and data-type as *m*.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If input *m*'s rank < 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.triu(np.ones((3, 3)))
>>> print(output)
[[1. 1. 1.]
[0. 1. 1.]
[0. 0. 1.]]
```

13.1.41 mindspore.numpy.triu_indices

mindspore.numpy.**triu_indices**(*n*, *k=0*, *m=None*)

Returns the indices for the upper-triangle of an (n, m) array.

Parameters

- **n** (*int*) –The size of the arrays for which the returned indices will be valid.
- **k** (*int, optional*) –Diagonal offset, default: 0 .

- **m** (*int, optional*) –The column dimension of the arrays for which the returned arrays will be valid. By default *m* is taken equal to *n*. Default: `None` .

Returns

The indices for the triangle. The returned tuple contains two tensors, each with the indices along one dimension of the tensor.

Raises

TypeError –If *n*, *k*, *m* are not numbers.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.triu_indices(3))
(Tensor(shape=[6], dtype=Int32, value= [0, 0, 0, 1, 1, 2]),
 Tensor(shape=[6], dtype=Int32, value= [0, 1, 2, 1, 2, 2]))
```

13.1.42 mindspore.numpy.triu_indices_from

mindspore.numpy.**triu_indices_from**(*arr*, *k=0*)

Returns the indices for the upper-triangle of *arr*.

Parameters

- **arr** (*Union[Tensor, list, tuple]*) –2-dimensional array.
- **k** (*int, optional*) –Diagonal offset, default: `0` .

Returns

triu_indices_from, tuple of 2 tensor, shape(N) Indices for the upper-triangle of *arr*.

Raises

- **TypeError** –If *arr* cannot be converted to tensor, or *k* is not a number.
- **ValueError** –If *arr* cannot be converted to a 2-dimensional tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> tensor = np.ones((3,3))
>>> print(np.triu_indices_from(tensor))
(Tensor(shape=[6], dtype=Int32, value= [0, 0, 0, 1, 1, 2]),
Tensor(shape=[6], dtype=Int32, value= [0, 1, 2, 1, 2, 2]))
```

13.1.43 mindspore.numpy.vander

mindspore.numpy.**vander**(*x*, *N=None*, *increasing=False*)

Generates a Vandermonde matrix.

The columns of the output matrix are powers of the input vector. The order of the powers is determined by the increasing boolean argument. Specifically, when increasing is *False*, the i-th output column is the input vector raised element-wise to the power of $N - i - 1$. Such a matrix with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

Parameters

- **x** (*Union[list, tuple, Tensor]*) –1-D input array.
- **N** (*int, optional*) –Number of columns in the output. If N is not specified, a square array is returned (`N = len(x)`).
- **increasing** (*bool, optional*) –Order of the powers of the columns. If True, the powers increase from left to right, if False (the default) they are reversed.

Returns

Vandermonde matrix. If *increasing* is *False*, the first column is $x^{(N-1)}$, the second $x^{(N-2)}$ and so forth. If *increasing* is *True*, the columns are $x^0, x^1, ..., x^{(N-1)}$.

Raises

- **TypeError** –If inputs have types not specified above.
- **ValueError** –If x is not 1-D, or $N < 0$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.vander([1., 2., 3., 4., 5.]))
[[  1.   1.   1.   1.   1.]
 [ 16.   8.   4.   2.   1.]
 [ 81.  27.   9.   3.   1.]
 [256.  64.  16.   4.   1.]
 [625. 125.  25.   5.   1.]]
```

13.1.44 mindspore.numpy.zeros

mindspore.numpy.**zeros**(*shape*, *dtype=mstype.float32*)

Returns a new tensor of given shape and type, filled with zeros.

Parameters

- **shape** (*Union[int, tuple, list]*) –the shape of the new tensor.
- **dtype** (Union[*mindspore.dtype*, str], optional) –Designated tensor dtype. Default is mstype.float32.

Returns

Tensor, with the designated *shape* and *dtype*, filled with zeros.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If *shape* entries have values < 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.zeros((2,2)))
[[0. 0.]
[0. 0.]]
```

13.1.45 mindspore.numpy.zeros_like

mindspore.numpy.**zeros_like**(*a*, *dtype=None*, *shape=None*)

Returns an array of zeros with the same shape and type as a given array.

Note: Input array must have the same size across a dimension. If *a* is not a Tensor, dtype is float32 by default if not provided.

Parameters

- **a** (*Union[Tensor, list, tuple]*) –The shape and data-type of a define these same attributes of the returned array.
- **dtype** (*mindspore.dtype*, optional) –Overrides the data type of the result.
- **shape** (*int, sequence of ints, optional*) –Overrides the shape of the result.

Returns

Tensor, array of zeros with the same shape and type as *a*.

Raises

ValueError –If *a* is not a Tensor, list or tuple.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((4,1,2))
>>> output = np.zeros_like(a)
>>> print(output)
[[[0. 0.]]
[[0. 0.]]
[[0. 0.]]
[[0. 0.]]]
```

13.2 Array Operation

Array operations focus on tensor manipulation.

- Manipulate the shape of the tensor

  For example, transpose a matrix:

```
input_x = np.arange(10).reshape(5, 2)
output = np.transpose(input_x)
print(output)
```

  The result is as follows:

```
[[0 2 4 6 8]
 [1 3 5 7 9]]
```

  Another example, swap two axes:

```
input_x = np.ones((1, 2, 3))
output = np.swapaxes(input_x, 0, 1)
print(output.shape)
```

  The result is as follows:

```
(2, 1, 3)
```

- Tensor splitting

  Divide the input tensor into multiple tensors equally, for example:

```
input_x = np.arange(9)
output = np.split(input_x, 3)
print(output)
```

  The result is as follows:

```
(Tensor(shape=[3], dtype=Int32, value= [0, 1, 2]), Tensor(shape=[3], dtype=Int32, value= [3,␣
↪4, 5]), Tensor(shape=[3], dtype=Int32, value= [6, 7, 8]))
```

- Tensor combination

  Concatenate the two tensors according to the specified axis, for example:

```
input_x = np.arange(0, 5)
input_y = np.arange(10, 15)
output = np.concatenate((input_x, input_y), axis=0)
print(output)
```

  The result is as follows:

```
[ 0  1  2  3  4 10 11 12 13 14]
```

API Name	Description	Supported Platforms
mindspore.numpy.append	Appends values to the end of a tensor.	Ascend GPU CPU
mindspore.numpy.apply_along_axis	Applies a function to 1-D slices along the given axis.	Ascend GPU CPU
mindspore.numpy.apply_over_axes	Applies a function repeatedly over multiple axes.	Ascend GPU CPU

continues on next page

Table 2 – continued from previous page

mindspore.numpy.argwhere	Find the indices of Tensor elements that are non-zero, grouped by element.	Ascend GPU CPU
mindspore.numpy.array_split	Splits a tensor into multiple sub-tensors.	Ascend GPU CPU
mindspore.numpy.array_str	Returns a string representation of the data in an array.	Ascend GPU CPU
mindspore.numpy.atleast_1d	Converts inputs to arrays with at least one dimension.	Ascend GPU CPU
mindspore.numpy.atleast_2d	Reshapes inputs as arrays with at least two dimensions.	Ascend GPU CPU
mindspore.numpy.atleast_3d	Reshapes inputs as arrays with at least three dimensions.	Ascend GPU CPU
mindspore.numpy.broadcast_arrays	Broadcasts any number of arrays against each other.	Ascend GPU CPU
mindspore.numpy.broadcast_to	Broadcasts an array to a new shape.	Ascend GPU CPU
mindspore.numpy.choose	Construct an array from an index array and a list of arrays to choose from.	Ascend GPU CPU
mindspore.numpy.column_stack	Stacks 1-D tensors as columns into a 2-D tensor.	Ascend GPU CPU
mindspore.numpy.concatenate	Joins a sequence of tensors along an existing axis.	Ascend GPU CPU
mindspore.numpy.dsplit	Splits a tensor into multiple sub-tensors along the 3rd axis (depth).	Ascend GPU CPU
mindspore.numpy.dstack	Stacks tensors in sequence depth wise (along the third axis).	Ascend GPU CPU
mindspore.numpy.expand_dims	Expands the shape of a tensor.	Ascend GPU CPU
mindspore.numpy.flip	Reverses the order of elements in an array along the given axis.	GPU CPU
mindspore.numpy.fliplr	Flips the entries in each row in the left/right direction.	GPU CPU
mindspore.numpy.flipud	Flips the entries in each column in the up/down direction.	GPU CPU
mindspore.numpy.hsplit	Splits a tensor into multiple sub-tensors horizontally (column-wise).	Ascend GPU CPU
mindspore.numpy.hstack	Stacks tensors in sequence horizontally.	Ascend GPU CPU
mindspore.numpy.intersect1d	Find the intersection of two Tensors.	Ascend GPU CPU
mindspore.numpy.moveaxis	Moves axes of an array to new positions.	Ascend GPU CPU
mindspore.numpy.piecewise	Evaluates a piecewise-defined function.	Ascend GPU CPU
mindspore.numpy.ravel	Returns a contiguous flattened tensor.	Ascend GPU CPU
mindspore.numpy.repeat	Repeats elements of an array.	Ascend GPU CPU
mindspore.numpy.reshape	Reshapes a tensor without changing its data.	Ascend GPU CPU
mindspore.numpy.roll	Rolls a tensor along given axes.	Ascend GPU CPU
mindspore.numpy.rollaxis	Rolls the specified axis backwards, until it lies in the given position.	Ascend GPU CPU
mindspore.numpy.rot90	Rotates a tensor by 90 degrees in the plane specified by axes.	GPU
mindspore.numpy.select	Returns an array drawn from elements in *choicelist*, depending on conditions.	Ascend GPU CPU
mindspore.numpy.setdiff1d	Find the set difference of two Tensors.	Ascend GPU CPU
mindspore.numpy.size	Returns the number of elements along a given axis.	Ascend GPU CPU
mindspore.numpy.split	Splits a tensor into multiple sub-tensors along the given axis.	Ascend GPU CPU
mindspore.numpy.squeeze	Return the Tensor after deleting the dimension of size 1 in the specified *axis*.	Ascend GPU CPU
mindspore.numpy.stack	Joins a sequence of arrays along a new axis.	Ascend GPU CPU

continues on next page

Table 2 – continued from previous page

mindspore.numpy.swapaxes	Interchanges two axes of a tensor.	Ascend GPU CPU
mindspore.numpy.take	Takes elements from an array along an axis.	Ascend GPU CPU
mindspore.numpy.take_along_axis	Takes values from the input array by matching 1d index and data slices.	Ascend GPU CPU
mindspore.numpy.tile	Constructs an array by repeating *a* the number of times given by *reps*.	Ascend GPU CPU
mindspore.numpy.transpose	Reverses or permutes the axes of a tensor; returns the modified tensor.	Ascend GPU CPU
mindspore.numpy.unique	Finds the unique elements of a tensor.	Ascend GPU CPU
mindspore.numpy.unravel_index	Converts a flat index or array of flat indices into a tuple of coordinate arrays.	Ascend GPU CPU
mindspore.numpy.vsplit	Splits a tensor into multiple sub-tensors vertically (row-wise).	Ascend GPU CPU
mindspore.numpy.vstack	Stacks tensors in sequence vertically.	Ascend GPU CPU
mindspore.numpy.where	Returns elements chosen from *x* or *y* depending on *condition*.	Ascend GPU CPU

13.2.1 mindspore.numpy.append

mindspore.numpy.**append**(*arr*, *values*, *axis=None*)

Appends values to the end of a tensor.

Parameters

- **arr** (Tensor) –Values are appended to a copy of this tensor.
- **values** (Tensor) –These values are appended to a copy of *arr*. It must be of the correct shape (the same shape as *arr*, excluding *axis*). If *axis* is not specified, *values* can be any shape and will be flattened before use.
- **axis** (*int, optional*) –The *axis* along which values are appended. If *axis* is not given, both *arr* and *values* are flattened before use, default is None.

Returns

Tensor, a copy of tensor with values appended to axis.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If specified axis exceeds *arr.ndim*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((2, 3))
>>> b = np.ones((2, 1))
>>> print(np.append(a, b, axis=1).shape)
(2, 4)
```

13.2.2 mindspore.numpy.apply_along_axis

mindspore.numpy.**apply_along_axis**(*func1d*, *axis*, *arr*, **args*, ***kwargs*)

Applies a function to 1-D slices along the given axis. Executes `func1d(a, *args, **kwargs)` where *func1d* operates on 1-D arrays and *a* is a 1-D slice of arr along axis.

Parameters

- **func1d** (*function*) –Maps *(M,) -> (Nj…)*. This function should accept 1-D arrays. It is applied to 1-D slices of arr along the specified axis.
- **axis** (*int*) –Axis along which arr is sliced.
- **arr** (Tensor) –Input array with shape *(Ni…, M, Nk…)*.
- **args** (*any*) –Additional arguments to *func1d*.
- **kwargs** (*any*) –Additional named arguments to *func1d*.

Returns

Tensor with shape *(Ni…, Nj…, Nk…)*, the output array. Its shape is identical to the shape of *arr*, except along the *axis* dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of *func1d*. So if *func1d* returns a scalar, the output will have one fewer dimensions than *arr*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

ValueError –If axis is out of the range.

Examples

```
>>> import mindspore.numpy as np
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> print(np.apply_along_axis(np.diag, -1, b))
[[[1 0 0]
[0 2 0]
[0 0 3]]
[[4 0 0]
[0 5 0]
[0 0 6]]
[[7 0 0]
[0 8 0]
[0 0 9]]]
```

13.2.3 mindspore.numpy.apply_over_axes

mindspore.numpy.**apply_over_axes**(*func*, *a*, *axes*)

Applies a function repeatedly over multiple axes.

func is called as *res* = *func(a, axis)*, where *axis* is the first element of *axes*. The result *res* of the function call must have either the same dimensions as *a* or one less dimension. If *res* has one less dimension than *a*, a dimension is inserted before *axis*. The call to *func* is then repeated for each axis in *axes*, with *res* as the first argument.

Parameters

- **func** (*function*) – This function must take two arguments, *func(a, axis)*.
- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input tensor.
- **axes** (*Union[int, list, tuple]*) – Axes over which *func* is applied; the elements must be integers.

Returns

Tensor. The number of dimensions is the same as *a*, but the shape can be different. This depends on whether *func* changes the shape of its output with respect to its input.

Raises

- **TypeError** – If input *a* is not array_like or *axes* is not int or sequence of ints.
- **ValueError** – If any axis is out of range or duplicate axes exist.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(10).reshape(2, 5).astype('float32')
>>> print(x)
[[0. 1. 2. 3. 4.]
 [5. 6. 7. 8. 9.]]
>>> print(np.apply_over_axes(np.sum, x, axes=0))
[[ 5.  7.  9. 11. 13.]]
```

13.2.4 mindspore.numpy.argwhere

mindspore.numpy.**argwhere**(*a*)

Find the indices of Tensor elements that are non-zero, grouped by element.

Parameters

a (*Union[list, tuple, Tensor]*) – Input tensor.

Returns

Tensor. Indices of elements that are non-zero. Indices are grouped by element. This Tensor will have shape $(N, a.ndim)$ where N is the number of non-zero items.

Raises

- **TypeError** – If input *a* is not array_like.
- **ValueError** – If dim of *a* equals to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([[[1, 0], [-5, 0]]])
>>> np.argwhere(x)
Tensor(shape=[2, 3], dtype=Int64, value=[[0, 0, 0], [0, 1, 0]])
```

13.2.5 mindspore.numpy.array_split

`mindspore.numpy.`**`array_split`**(*x*, *indices_or_sections*, *axis=0*)

Splits a tensor into multiple sub-tensors.

Note: Currently, array_split only supports `mindspore.float32` on `CPU`.

The only difference between `np.split` and `np.array_split` is that `np.array_split` allows indices_or_sections to be an integer that does not equally divide the axis. For a tensor of length l that should be split into n sections, it returns $l\%n$ sub-arrays of size $l//n+1$ and the rest of size $l//n$.

Parameters

- **`x`** (`Tensor`) –A Tensor to be divided.
- **`indices_or_sections`** (*`Union[int, tuple(int), list(int)]`*) –If integer, N, the tensor will be divided into N tensors along axis. If tuple(int), list(int) or of sorted integers, the entries indicate where along axis the array is split. For example, $[2, 3]$ would, for $axis = 0$, result in three sub-tensors $x[: 2]$, $x[2 : 3]$ and $x[3 :]$. If an index exceeds the dimension of the array along axis, an empty sub-array is returned correspondingly.
- **`axis`** (*`int`*) –The axis along which to split. Default: `0` .

Returns

A list of sub-tensors.

Raises

- **`TypeError`** –If argument *indices_or_sections* is not integer, tuple(int) or list(int) or argument *axis* is not integer.
- **`ValueError`** –If argument *axis* is out of range of $[-x.ndim, x.ndim)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.arange(9).astype("float32")
>>> output = np.array_split(input_x, 4)
>>> print(output)
(Tensor(shape=[3], dtype=Float32,
    value= [ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00]),
Tensor(shape=[2], dtype=Float32,
    value= [ 3.00000000e+00,  4.00000000e+00]),
Tensor(shape=[2], dtype=Float32,
    value= [ 5.00000000e+00,  6.00000000e+00]),
Tensor(shape=[2], dtype=Float32,
    value= [ 7.00000000e+00,  8.00000000e+00]))
```

13.2.6 mindspore.numpy.array_str

mindspore.numpy.**array_str**(*a*)

Returns a string representation of the data in an array.

The data in the array is returned as a single string. This function is similar to array_repr, the difference being that array_repr also returns information on the kind of array and its data type.

Note: Numpy argument *max_line_width*, *precision* and *suppress_small* are not supported. Graph mode does not support the function.

Parameters

a (Tensor) – Input data.

Returns

String.

Supported Platforms:

Ascend GPU CPU

Raises

TypeError – If input is not tensor.

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5)
>>> np.array_str(x)
'[0 1 2 3 4]'
```

13.2.7 mindspore.numpy.atleast_1d

mindspore.numpy.**atleast_1d**(**arys*)

Converts inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

Note: In graph mode, returns a tuple of tensor instead of a list of tensors.

Parameters

***arys** (Tensor) – one or more input tensors.

Returns

Tensor, or list of tensors, each with `a.ndim >= 1`.

Raises

TypeError – If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((2, 3))
>>> b = np.ones(())
>>> c = np.ones(5)
>>> output = np.atleast_1d(a, b, c)
>>> print(output)
    [Tensor(shape=[2, 3], dtype=Float32, value=
    [[1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
    [1.00000000e+00, 1.00000000e+00, 1.00000000e+00]]),
    Tensor(shape=[1], dtype=Float32, value= [1.00000000e+00]),
    Tensor(shape=[5], dtype=Float32,
    value= [1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
    1.00000000e+00, 1.00000000e+00])]
```

13.2.8 mindspore.numpy.atleast_2d

mindspore.numpy.**atleast_2d**(**arys*)

Reshapes inputs as arrays with at least two dimensions.

Note: In graph mode, returns a tuple of tensor instead of a list of tensors.

Parameters

***arys** (Tensor) – one or more input tensors.

Returns

Tensor, or list of tensors, each with `a.ndim >= 2`.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((2, 3))
>>> b = np.ones(())
>>> c = np.ones(5)
>>> output = np.atleast_2d(a, b, c)
>>> print(output)
    [Tensor(shape=[2, 3], dtype=Float32, value=
    [[1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
    [1.00000000e+00, 1.00000000e+00, 1.00000000e+00]]),
    Tensor(shape=[1, 1], dtype=Float32, value= [[1.00000000e+00]]),
    Tensor(shape=[1, 5], dtype=Float32,
    value= [[1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
    1.00000000e+00, 1.00000000e+00]])]
```

13.2.9 mindspore.numpy.atleast_3d

mindspore.numpy.**atleast_3d**(**arys*)

Reshapes inputs as arrays with at least three dimensions.

Note: In graph mode, returns a tuple of tensor instead of a list of tensors.

Parameters

***arys** (Tensor) –one or more input tensors.

Returns

Tensor, or list of tensors, each with `a.ndim >= 3`. For example, a 1-D array of shape *(N,)* becomes a tensor of shape *(1, N, 1)*, and a 2-D array of shape *(M, N)* becomes a tensor of shape *(M, N, 1)*.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((2, 3))
>>> b = np.ones(())
>>> c = np.ones(5)
>>> output = np.atleast_3d(a, b, c)
>>> print(output)
    [Tensor(shape=[2, 3, 1], dtype=Float32, value=
    [[[1.00000000e+00], [1.00000000e+00], [1.00000000e+00]],
    [[1.00000000e+00], [1.00000000e+00], [1.00000000e+00]]]),
    Tensor(shape=[1, 1, 1], dtype=Float32, value= [[[1.00000000e+00]]]),
    Tensor(shape=[1, 5, 1], dtype=Float32,
    value= [[[1.00000000e+00], [1.00000000e+00], [1.00000000e+00],
    [1.00000000e+00], [1.00000000e+00]]])]
```

13.2.10 mindspore.numpy.broadcast_arrays

mindspore.numpy.**broadcast_arrays**(**args*)

Broadcasts any number of arrays against each other.

Note: Numpy argument *subok* is not supported. In graph mode, returns a tuple of Tensor instead of a list of Tensor.

Parameters

***args** (Tensor) – The arrays to broadcast.

Returns

List of Tensor.

Raises

ValueError – If arrays cannot be broadcast.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([[1,2,3]])
>>> y = np.array([[4],[5]])
>>> output = np.broadcast_arrays(x, y)
>>> print(output)
[Tensor(shape=[2, 3], dtype=Int32, value=
[[1, 2, 3],
[1, 2, 3]]), Tensor(shape=[2, 3], dtype=Int32, value=
[[4, 4, 4],
[5, 5, 5]])]
```

13.2.11 mindspore.numpy.broadcast_to

mindspore.numpy.**broadcast_to**(*array*, *shape*)

Broadcasts an array to a new shape.

Parameters

- **array** (Tensor) –The array to broadcast.
- **shape** (*tuple*) –The shape of the desired array.

Returns

Tensor, original array broadcast to the given shape.

Raises

ValueError –If array cannot be broadcast to shape.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1, 2, 3])
>>> output = np.broadcast_to(x, (3, 3))
>>> print(output)
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

13.2.12 mindspore.numpy.choose

mindspore.numpy.**choose**(*a*, *choices*, *mode='clip'*)

Construct an array from an index array and a list of arrays to choose from. Given an "index" array *a* of integers and a sequence of n arrays (choices), *a* and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these *Ba* and *Bchoices[i], i = 0,⋯,n-1* we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each *i*. Then, a new array with `shape Ba.shape` is created as follows:

- if `mode='raise'` (the default), then, first of all, each element of *a* (and thus *Ba*) must be in the range *[0, n-1]*; now, suppose that *i* (in that range) is the value at the *(j0, j1, ⋯, jm)* position in *Ba* - then the value at the same position in the new array is the value in `Bchoices[i]` at that same position;
- if `mode='wrap'`, values in *a* (and thus *Ba*) may be any (signed) integer; modular arithmetic is used to map integers outside the range `[0, n-1]` back into that range; and then the new array is constructed as above;
- if `mode='clip'`, values in *a* (and thus *Ba*) may be any (signed) integer; negative integers are mapped to 0; values greater than *n-1* are mapped to *n-1*; and then the new array is constructed as above.

Note: Numpy argument *out* is not supported. `mode = 'raise'` is not supported, and the default mode is 'clip' instead.

Parameters

- **a** (*int array*) –This array must contain integers in `[0, n-1]`, where *n* is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.
- **choices** (*sequence of arrays*) –Choice arrays. *a* and all of the *choices* must be broadcastable to the same shape. If *choices* is itself an array, then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the "sequence".
- **mode** (*'raise', 'wrap', 'clip', optional*) –Specifies how indices outside `[0, n-1]` will be treated:

  'raise' - raise an error;

  'wrap' - wrap around;

  'clip' - clip to the range. 'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns

Tensor, the merged result.

Raises

ValueError –If *a* and any of the *choices* cannot be broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23], [30, 31, 32, 33]]
>>> print(np.choose([2, 3, 1, 0], choices))
[20 31 12  3]
>>> print(np.choose([2, 4, 1, 0], choices, mode='clip'))
[20 31 12  3]
>>> print(np.choose([2, 4, 1, 0], choices, mode='wrap'))
[20  1 12  3]
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> print(np.choose(a, choices))
[[ 10 -10  10]
 [-10  10 -10]
 [ 10 -10  10]]
```

13.2.13 mindspore.numpy.column_stack

`mindspore.numpy.`**column_stack**(*tup*)

Stacks 1-D tensors as columns into a 2-D tensor. 2-D tensors are stacked as-is, like np.hstack.

Parameters

tup (*Union[Tensor, tuple, list]*) –A sequence of 1-D or 2-D tensors. All of them must have the same shape except the axis to be concatenated.

Returns

2-D Tensor, formed by stacking the given tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

- **TypeError** –If *tup* is not Tensor, list or tuple.
- **ValueError** –If *tup* is empty.

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([1, 2, 3]).astype('int32')
>>> x2 = np.array([4, 5, 6]).astype('int32')
>>> output = np.column_stack((x1, x2))
>>> print(output)
[[1 4]
 [2 5]
 [3 6]]
```

13.2.14 mindspore.numpy.concatenate

`mindspore.numpy.`**concatenate**(*arrays*, *axis=0*)

Joins a sequence of tensors along an existing axis.

Note: To match Numpy behaviour, *axis* >= 32 will not cause value error, the *axis* will be treated as `None` instead.

Parameters

- **arrays** (*Union[Tensor, tuple(Tensor), list(Tensor)]*) –a tensor or a list of tensors to be concatenated.
- **axis** (*Union[None, int], optional*) –The axis along which the tensors will be joined, if *axis* is `None`, tensors are flattened before use. Default: `0` .

Returns

A tensor concatenated from a tensor or a list of tensors.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If *axis* is not in the range of $[-ndim, ndim - 1]$, and less than 32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.ones((1,2,3))
>>> x2 = np.ones((1,2,1))
>>> x = np.concatenate((x1, x2), axis=-1)
>>> print(x.shape)
(1, 2, 4)
```

13.2.15 mindspore.numpy.dsplit

mindspore.numpy.**dsplit**(*x*, *indices_or_sections*)

Splits a tensor into multiple sub-tensors along the 3rd axis (depth). It is equivalent to split with $axis = 2$ (default), the array is always split along the third axis regardless of the array dimension.

Parameters

- **x** (Tensor) –A Tensor to be divided.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –If integer, N, the tensor will be divided into N equal tensors along axis. If tuple(int), list(int) or of sorted integers, the entries indicate where along axis the array is split. For example, $[2, 3]$ would, for $axis = 0$, result in three sub-tensors $x[: 2]$, $x[2 : 3]$ and $x[3 :]$. If an index exceeds the dimension of the array along axis, an empty sub-array is returned correspondingly.

Returns

A list of sub-tensors.

Raises

TypeError –If argument *indices_or_sections* is not integer.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.arange(6).reshape((1, 2, 3)).astype('float32')
>>> output = np.dsplit(input_x, 3)
>>> print(output)
(Tensor(shape=[1, 2, 1], dtype=Float32,
value=[[[ 0.00000000e+00],
        [ 3.00000000e+00]]]),
Tensor(shape=[1, 2, 1], dtype=Float32,
value=[[[ 1.00000000e+00],
        [ 4.00000000e+00]]]),
Tensor(shape=[1, 2, 1], dtype=Float32,
value=[[[ 2.00000000e+00],
        [ 5.00000000e+00]]]))
```

13.2.16 mindspore.numpy.dstack

mindspore.numpy.**dstack**(*tup*)

Stacks tensors in sequence depth wise (along the third axis). This is equivalent to concatenation along the third axis. 1-D tensors $(N,)$ should be reshaped to $(1, N, 1)$. 2-D tensors (M, N) should be reshaped to $(M, N, 1)$ before concatenation.

Parameters

tup (*Union[Tensor, tuple, list]*) –A sequence of tensors. The tensors must have the same shape along all but the third axis. 1-D or 2-D tensors must have the same shape.

Returns

Stacked Tensor, formed by stacking the given tensors.

Supported Platforms:

Ascend GPU CPU

Raises

- **TypeError** –If *tup* is not Tensor, list or tuple.
- **ValueError** –If *tup* is empty.

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([1, 2, 3]).astype('float32')
>>> x2 = np.array([4, 5, 6]).astype('float32')
>>> output = np.dstack((x1, x2))
>>> print(output)
[[[1. 4.]
  [2. 5.]
  [3. 6.]]]
```

13.2.17 mindspore.numpy.expand_dims

mindspore.numpy.**expand_dims**(*a*, *axis*)

Expands the shape of a tensor.

Inserts a new axis that will appear at the axis position in the expanded tensor shape.

Parameters

- **a** (Tensor) –Input tensor array.
- **axis** (*Union[int, list(int), tuple(int)]*) –Position in the expanded axes where the new axis is placed,

Returns

Tensor, with the number of dimensions increased at specified axis.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If axis exceeds a.ndim.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((2,2))
>>> x = np.expand_dims(x,0)
>>> print(x.shape)
(1, 2, 2)
```

13.2.18 mindspore.numpy.flip

mindspore.numpy.**flip**(*m*, *axis=None*)

Reverses the order of elements in an array along the given axis.

The shape of the array is preserved, but the elements are reordered.

Parameters

- **m** (Tensor) –Input array.
- **axis** (*Union[int, tuple(int), None], optional*) –Axis or axes along which to flip over. The default, axis=None, will flip over all of the axes of the input array. If *axis* is negative it counts from the last to the first axis. If *axis* is a tuple of integers, flipping is performed on all of the axes specified in the tuple.

Returns

Tensor, with the entries of *axis* reversed.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> A = np.arange(8.0).reshape((2,2,2))
>>> output = np.flip(A)
>>> print(output)
[[[7. 6.]
  [5. 4.]]
 [[3. 2.]
  [1. 0.]]]
>>> output = np.flip(A, (0, 2))
>>> print(output)
[[[5. 4.]
  [7. 6.]]
 [[1. 0.]
  [3. 2.]]]
```

13.2.19 mindspore.numpy.fliplr

mindspore.numpy.**fliplr**(*m*)

Flips the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

Parameters

m (Tensor) –Input array.

Returns

Tensor.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> A = np.arange(8.0).reshape((2,2,2))
>>> output = np.fliplr(A)
>>> print(output)
[[[2. 3.]
  [0. 1.]]
 [[6. 7.]
  [4. 5.]]]
```

13.2.20 mindspore.numpy.flipud

mindspore.numpy.**flipud**(*m*)

Flips the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

Parameters

m (Tensor) –Input array.

Returns

Tensor.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> A = np.arange(8.0).reshape((2,2,2))
>>> output = np.flipud(A)
>>> print(output)
[[[4. 5.]
  [6. 7.]]
 [[0. 1.]
  [2. 3.]]]
```

13.2.21 mindspore.numpy.hsplit

mindspore.numpy.**hsplit**(*x*, *indices_or_sections*)

Splits a tensor into multiple sub-tensors horizontally (column-wise). It is equivalent to split with $axis = 1$ (default), the array is always split along the second axis regardless of the array dimension.

Parameters

- **x** (Tensor) –A Tensor to be divided.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –If integer, *N*, the tensor will be divided into *N* equal tensors along axis. If tuple(int), list(int) or of sorted integers, the entries indicate where along axis the array is split. For example, $[2, 3]$ would, for $axis = 0$, result in three sub-tensors $x[: 2]$, $x[2 : 3]$ and $x[3 :]$. If an index exceeds the dimension of the array along axis, an empty sub-array is returned correspondingly.

Returns

A list of sub-tensors.

Raises

TypeError –If argument *indices_or_sections* is not integer.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.arange(6).reshape((2, 3)).astype('float32')
>>> output = np.hsplit(input_x, 3)
>>> print(output)
(Tensor(shape=[2, 1], dtype=Float32,
value=[[ 0.00000000e+00],
       [ 3.00000000e+00]]),
Tensor(shape=[2, 1], dtype=Float32,
value=[[ 1.00000000e+00],
       [ 4.00000000e+00]]),
Tensor(shape=[2, 1], dtype=Float32,
value=[[ 2.00000000e+00],
       [ 5.00000000e+00]]))
```

13.2.22 mindspore.numpy.hstack

mindspore.numpy.**hstack**(*tup*)

Stacks tensors in sequence horizontally. This is equivalent to concatenation along the second axis, except for 1-D tensors where it concatenates along the first axis.

Parameters

tup (*Union[Tensor, tuple, list]*) – A sequence of 1-D or 2-D tensors. The tensors must have the same shape along all but the second axis, except 1-D tensors which can be any length.

Returns

Stacked Tensor, formed by stacking the given tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

- **TypeError** – If *tup* is not Tensor, list or tuple.
- **ValueError** – If *tup* is empty.

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([1, 2, 3]).astype('float32')
>>> x2 = np.array([4, 5, 6]).astype('float32')
>>> output = np.hstack((x1, x2))
>>> print(output)
[1. 2. 3. 4. 5. 6.]
```

13.2.23 mindspore.numpy.intersect1d

mindspore.numpy.**intersect1d**(*ar1*, *ar2*, *assume_unique=False*, *return_indices=False*)

Find the intersection of two Tensors. Return the sorted, unique values that are in both of the input Tensors.

Parameters

- **ar1** (*Union[int, float, bool, list, tuple, Tensor]*) – Input tensor.
- **ar2** (*Union[int, float, bool, list, tuple, Tensor]*) – Input tensor.
- **assume_unique** (*bool*) – If *True*, the input Tensors are assumed to be unique, which can speed up the calculation. If *True* but *ar1* or *ar2* are not unique, incorrect results and out-of-bounds indices could result. Default: `False`.
- **return_indices** (*bool*) – If *True*, the indices which correspond to the intersection of two Tensors are returned. The first instance of a value is used if there are multiple. Default: `False`.

Returns

Tensor or tuple of Tensors. If *return_indices* is `False`, return the intersection tensor, otherwise return tuple of tensors.

Raises

- **TypeError** – If input *ar1* or *ar2* is not array_like.
- **TypeError** – If *assume_unique* or *return_indices* is not bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
Tensor(shape=[2], dtype=Int32, value=[1, 3])
```

13.2.24 mindspore.numpy.moveaxis

`mindspore.numpy.`**`moveaxis`**(*a*, *source*, *destination*)

Moves axes of an array to new positions.

Other axes remain in their original order.

Parameters

- **a** (`Tensor`) –The array whose axes should be reordered.
- **source** (*int, sequence of ints*) –Original positions of the axes to move. These must be unique.
- **destination** (*int, sequence of ints*) –Destination positions for each of the original axes. These must also be unique.

Returns

Tensor, array with moved axes.

Raises

ValueError –If axes are out of the range of `[-a.ndim, a.ndim)`, or if the axes contain duplicates.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.zeros((3, 4, 5))
>>> output = np.moveaxis(x, 0, -1)
>>> print(output.shape)
(4, 5, 3)
>>> output = np.moveaxis(x, -1, 0)
>>> print(output.shape)
(5, 3, 4)
>>> output = np.moveaxis(x, [0, 1, 2], [-1, -2, -3])
>>> print(output.shape)
(5, 4, 3)
```

13.2.25 mindspore.numpy.piecewise

mindspore.numpy.**piecewise**(*x*, *condlist*, *funclist*, **args*, ***kw*)

Evaluates a piecewise-defined function. Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

Parameters

- **x** (*Union[int, float, bool, list, tuple, Tensor]*) –The input domain.
- **condlist** (*Union[bool, list[Tensor, bool]]*) –Each boolean array corresponds to a function in *funclist*. Wherever *condlist[i]* is True, *funclist[i](x)* is used as the output value. Each boolean array in *condlist* selects a piece of *x*, and should therefore be of the same shape as *x*. The length of *condlist* must correspond to that of *funclist*. If one extra function is given, i.e. if `len(funclist) == len(condlist) + 1`, then that extra function is the default value, used wherever all conditions are false.
- **funclist** (*Union[list[callables], list[scalars]]*) –Each function is evaluated over *x* wherever its corresponding condition is True. It should take a 1d array as input and give an 1d array or a scalar value as output. If, instead of a callable, a scalar is provided then a constant function `(lambda x: scalar)` is assumed.
- **args** (*any*) –Any further arguments given to *piecewise* are passed to the functions upon execution, i.e., if called `piecewise(..., ..., 1, 'a')`, then each function is called as `f(x, 1, 'a')`.
- **kw** (*any*) –Keyword arguments used in calling *piecewise* are passed to the functions upon execution, i.e., if called `piecewise(..., ..., alpha=1)`, then each function is called as `f(x, alpha=1)`.

Returns

Tensor, the output is the same shape and type as *x* and is found by calling the functions in *funclist* on the appropriate portions of *x*, as defined by the boolean arrays in *condlist*. Portions not covered by any condition have a default value of 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

ValueError –If length of *funclist* is not in `(len(condlist), len(condlist) + 1)`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.linspace(-2.5, 2.5, 6)
>>> print(np.piecewise(x, [x < 0, x >= 0], [-1, 1]))
[-1 -1 -1  1  1  1]
```

13.2.26 mindspore.numpy.ravel

mindspore.numpy.**ravel**(*x*)

Returns a contiguous flattened tensor.

A 1-D tensor, containing the elements of the input, is returned.

Parameters

x (`Tensor`) –A tensor to be flattened.

Returns

Flattened tensor, has the same data type as the original tensor *x*.

Raises

TypeError – If *x* is not tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((2,3,4))
>>> output = np.ravel(x)
>>> print(output.shape)
(24,)
```

13.2.27 mindspore.numpy.repeat

`mindspore.numpy.repeat`(*a*, *repeats*, *axis=None*)

Repeats elements of an array.

Parameters

- **a** (`Tensor`) – Input array.
- **repeats** (*int, sequence of ints*) – The number of repetitions for each element. *repeats* is broadcasted to fit the shape of the given axis.
- **axis** (*int, optional*) – The axis along which to repeat values. By default, use the flattened input array, and return a flat output array. Default: `None` .

Returns

Tensor, output array which has the same shape as *a*, except along the given axis.

Raises

- **ValueError** – If axis is out of range.
- **TypeError** – If input *a* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.repeat(np.array(3), 4)
>>> print(output)
[3 3 3 3]
>>> x = np.array([[1,2],[3,4]])
>>> output = np.repeat(x, 2)
>>> print(output)
[1 1 2 2 3 3 4 4]
```

(continues on next page)

(continued from previous page)

```
>>> output = np.repeat(x, 3, axis=1)
>>> print(output)
[[1 1 1 2 2 2]
[3 3 3 4 4 4]]
>>> output = np.repeat(x, [1, 2], axis=0)
>>> print(output)
[[1 2]
[3 4]
[3 4]]
```

13.2.28 mindspore.numpy.reshape

mindspore.numpy.**reshape**(*x*, *new_shape*)

Reshapes a tensor without changing its data.

Parameters

- **x** (Tensor) – A tensor to be reshaped.
- **new_shape** (*Union[int, list(int), tuple(int)]*) – The new shape should be compatible with the original shape. If the tuple has only one element, the result will be a 1-D tensor of that length. One shape dimension can be –1. In this case, the value is inferred from the length of the tensor and remaining dimensions.

Returns

Reshaped Tensor. Has the same data type as the original tensor *x*.

Raises

- **TypeError** – If new_shape is not integer, list or tuple, or *x* is not tensor.
- **ValueError** – If new_shape is not compatible with the original shape.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]])
>>> output = np.reshape(x, (3, 2))
>>> print(output)
[[-0.1  0.3]
 [ 3.6  0.4]
 [ 0.5 -3.2]]
>>> output = np.reshape(x, (3, -1))
>>> print(output)
[[-0.1  0.3]
 [ 3.6  0.4]
 [ 0.5 -3.2]]
>>> output = np.reshape(x, (6, ))
>>> print(output)
[-0.1  0.3  3.6  0.4  0.5 -3.2]
```

13.2.29 mindspore.numpy.roll

mindspore.numpy.**roll**(*a*, *shift*, *axis=None*)

Rolls a tensor along given axes.

Elements that rolls beyond the last position are re-introduced at the first.

Parameters

- **a** (Tensor) – Input tensor.
- **shift** (*Union[int, tuple(int)]*) – The number of places by which elements are shifted. If a tuple, then axis must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If shift is an int while axis is a tuple of integers, then the same value is used for all given axes.
- **axis** (*Union[int, tuple(int)], optional*) – Axis or axes along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored. Default: None .

Returns

Tensor, with the same shape as a.

Supported Platforms:

Ascend GPU CPU

Raises

- **TypeError** – If input arguments have types not specified above.
- **ValueError** – If axis exceeds *a.ndim*, or *shift* and *axis* cannot broadcast.

Examples

```
>>> import mindspore.numpy as np
>>> a = np.reshape(np.arange(12), (3, 4))
>>> print(np.roll(a, [2,-3], [0,-1]))
    [[ 7  4  5  6]
     [11  8  9 10]
     [ 3  0  1  2]]
```

13.2.30 mindspore.numpy.rollaxis

mindspore.numpy.**rollaxis**(*x*, *axis*, *start=0*)

Rolls the specified axis backwards, until it lies in the given position. The positions of the other axes do not change relative to one another.

Parameters

- **x** (Tensor) – A Tensor to be transposed.
- **axis** (*int*) – The axis to be rolled.
- **start** (*int*) – Default: 0 . If *start* <= *axis*, the axis is rolled back until it lies in this position (*start*). If *start* > *axis*: the axis is rolled until it lies before this position (*start*). If *start* < 0, the start will be normalized as a non-negative number (more details can be seen in the source code.)

Returns

Transposed Tensor. Has the same data type as the original tensor *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

- **TypeError** –If *axis* or *start* is not integer, or *x* is not tensor.
- **ValueError** –If *axis* is not in the range of $[-ndim, ndim-1]$ or *start* is not in the range of $[-ndim, ndim]$.

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((2,3,4))
>>> output = np.rollaxis(x, 0, 2)
>>> print(output.shape)
(3, 2, 4)
```

13.2.31 mindspore.numpy.rot90

mindspore.numpy.**rot90**(*a*, *k=1*, *axes=(0, 1)*)

Rotates a tensor by 90 degrees in the plane specified by axes. Rotation direction is from the first towards the second axis.

Parameters

- **a** (`Tensor`) –Input tensor of two or more dimensions.
- **k** (*int*) –Number of times the tensor is rotated by 90 degrees. Default: `1` .
- **axes** (*Union[tuple(int), list(int)], optional*) –The tensor is rotated in the plane defined by the axes. Default: `(0, 1)` . Axes must be different and with the shape of *(2,)*.

Returns

Tensor.

Raises

- **TypeError** –If input *a* is not a Tensor or the argument *k* is not integer or the argument *axes* is not tuple of integers or list of ints.
- **ValueError** –If any axis is out of range or the length of *axes* is not *2*.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(24).reshape((2, 3, 4))
>>> output = np.rot90(a)
>>> print(output)
[[[ 8  9 10 11]
  [20 21 22 23]]
 [[ 4  5  6  7]
  [16 17 18 19]]
```

(continues on next page)

(continued from previous page)

```
 [[ 0  1  2  3]
  [12 13 14 15]]]
>>> output = np.rot90(a, 3, (1, 2))
>>> print(output)
[[[ 8  4  0]
  [ 9  5  1]
  [10  6  2]
  [11  7  3]]
 [[20 16 12]
  [21 17 13]
  [22 18 14]
  [23 19 15]]]
```

13.2.32 mindspore.numpy.select

mindspore.numpy.**select**(*condlist*, *choicelist*, *default=0*)

Returns an array drawn from elements in *choicelist*, depending on conditions.

Parameters

- **condlist** (*Union[int, float, bool, list, tuple, Tensor]*) – The list of conditions which determine from which array in *choicelist* the output elements are taken. When multiple conditions are satisfied, the first one encountered in *condlist* is used.
- **choicelist** (*Union[int, float, bool, list, tuple, Tensor]*) – The list of arrays from which the output elements are taken. It has to be of the same length as *condlist*.
- **default** (*scalar, optional*) – The element inserted in output when all conditions evaluate to *False*. Default: 0 .

Returns

Tensor, the output at position *m* is the *m-th* element of the array in *choicelist* where the *m-th* element of the corresponding array in *condlist* is *True*.

Raises

ValueError – If len(condlist) != len(choicelist).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> condlist = [[True, True, True, False, False], [False, False, True, False, True]]
>>> choicelist = [[0, 1, 2, 3, 4], [0, 1, 4, 9, 16]]
>>> output = np.select(condlist, choicelist)
>>> print(output)
[ 0  1  2  0 16]
```

13.2.33 mindspore.numpy.setdiff1d

mindspore.numpy.**setdiff1d**(*ar1*, *ar2*, *assume_unique=False*)

Find the set difference of two Tensors. Return the unique values in *ar1* that are not in *ar2*.

Parameters

- **ar1** (*Union[int, float, bool, list, tuple, Tensor]*) –Input tensor.
- **ar2** (*Union[int, float, bool, list, tuple, Tensor]*) –Input tensor.
- **assume_unique** (*bool, optional*) –If *True*, the input Tensors are assumed to be unique, which can speed up the calculation. If *True* but *ar1* or *ar2* are not unique, incorrect results and out-of-bounds indices could result. Default: `False`.

Returns

1D Tensor of values in *ar1* that are not in *ar2*. The result is sorted when *assume_unique* = `False` , but otherwise only sorted if the input is sorted.

Raises

- **TypeError** –If input *ar1* or *ar2* is not array_like.
- **TypeError** –If *assume_unique* is not bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([1, 2, 3, 2, 4, 1])
>>> b = np.array([3, 4, 5, 6])
>>> np.setdiff1d(a, b)
Tensor(shape=[2], dtype=Int32, value=[1, 2])
```

13.2.34 mindspore.numpy.size

mindspore.numpy.**size**(*a*, *axis=None*)

Returns the number of elements along a given axis.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) –Input data.
- **axis** (*int, optional*) –Axis along which the elements are counted. Default: `None`. If None, give the total number of elements.

Returns

Number of elements along the specified axis.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

- **TypeError** –If input is not array_like or *axis* is not int.
- **ValueError** –If any axis is out of range or duplicate axes exist.

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(10).reshape(2, 5).astype('float32')
>>> print(np.size(x))
10
>>> print(np.size(x, axis=1))
5
```

13.2.35 mindspore.numpy.split

mindspore.numpy.**split**(*x*, *indices_or_sections*, *axis=0*)

Splits a tensor into multiple sub-tensors along the given axis.

Parameters

- **x** (Tensor) –A Tensor to be divided.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) –If integer, N, the tensor will be divided into N equal tensors along axis. If tuple(int), list(int) or of sorted integers, the entries indicate where along axis the array is split. For example, $[2, 3]$ would, for $axis = 0$, result in three sub-tensors $x[: 2]$, $x[2 : 3]$ and $x[3 :]$. If an index exceeds the dimension of the array along axis, an empty sub-array is returned correspondingly.
- **axis** (*int, optional*) –The axis along which to split. Default: 0 .

Returns

A tuple of sub-tensors.

Raises

- **TypeError** –If argument *indices_or_sections* is not integer, tuple(int) or list(int) or argument *axis* is not integer.
- **ValueError** –If argument *axis* is out of range of $[-x.ndim, x.ndim)$.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.arange(9).astype("float32")
>>> output = np.split(input_x, 3)
>>> print(output)
(Tensor(shape=[3], dtype=Float32,
  value= [ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00]),
 Tensor(shape=[3], dtype=Float32,
  value= [ 3.00000000e+00,  4.00000000e+00,  5.00000000e+00]),
 Tensor(shape=[3], dtype=Float32,
  value= [ 6.00000000e+00,  7.00000000e+00,  8.00000000e+00]))
```

13.2.36 mindspore.numpy.squeeze

mindspore.numpy.**squeeze**(*a*, *axis=None*)

Return the Tensor after deleting the dimension of size 1 in the specified *axis*.

If *axis* = *None*, it will remove all the dimensions of size 1. If *axis* is specified, it will remove the dimensions of size 1 in the given *axis*. For example, if the dimension is not specified *axis* = *None*, input shape is (A, 1, B, C, 1, D), then the shape of the output Tensor is (A, B, C, D). If the dimension is specified, the squeeze operation is only performed in the specified dimension. If input shape is (A, 1, B), when *axis* = 0 or *axis* = 2, the input tensor is not changed, while when *axis* = 1, the input tensor shape is changed to (A, B).

Parameters

- **a** (Tensor) –Input tensor array.
- **axis** (*Union[None, int, list(int), tuple(list)], optional*) –The axis(axes) to squeeze, default: None .

Returns

Tensor, with all or a subset of the dimensions of length 1 removed.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If specified axis has shape entry > 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((1,2,2,1))
>>> x = np.squeeze(x)
>>> print(x.shape)
(2, 2)
```

13.2.37 mindspore.numpy.stack

mindspore.numpy.**stack**(*arrays*, *axis=0*)

Joins a sequence of arrays along a new axis.

The *axis* parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension.

Note: Numpy argument out is not supported.

Parameters

- **arrays** (*sequence of Tensor*) –Each array must have the same shape.
- **axis** (*int, optional*) –The axis in the result array along which the input arrays are stacked. Default: 0 .

Returns

Tensor, The stacked array has one more dimension than the input arrays.

Raises

ValueError –If input is not Tensor, tuple, or list.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> arrays = [np.ones((3, 4)) for _ in range(10)]
>>> output = np.stack(arrays, axis=0)
>>> print(output.shape)
(10, 3, 4)
>>> output = np.stack(arrays, axis=1)
>>> print(output.shape)
(3, 10, 4)
>>> output = np.stack(arrays, axis=2)
>>> print(output.shape)
(3, 4, 10)
```

13.2.38 mindspore.numpy.swapaxes

mindspore.numpy.**swapaxes**(*x*, *axis1*, *axis2*)

Interchanges two axes of a tensor.

Parameters

- **x** (Tensor) –A tensor to be transposed.
- **axis1** (*int*) –First axis.
- **axis2** (*int*) –Second axis.

Returns

Transposed tensor, has the same data type as the original tensor *x*.

Raises

- **TypeError** –If *axis1* or *axis2* is not integer, or *x* is not tensor.
- **ValueError** –If *axis1* or *axis2* is not in the range of $[-ndim, ndim - 1]$.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((2,3,4))
>>> output = np.swapaxes(x, 0, 2)
>>> print(output.shape)
(4,3,2)
```

13.2.39 mindspore.numpy.take

mindspore.numpy.**take**(*a*, *indices*, *axis=None*, *mode='clip'*)

Takes elements from an array along an axis.

When axis is not None, this function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis. A call such as `np.take(arr, indices, axis=3)` is equivalent to `arr[:,:,:,indices,...]`.

Note: Numpy argument out is not supported. `mode = 'raise'` is not supported, and the default mode is 'clip' instead.

Parameters

- **a** (Tensor) –Source array with shape *(Ni…, M, Nk…)*.
- **indices** (Tensor) –The indices with shape *(Nj…)* of the values to extract.
- **axis** (*int, optional*) –The axis over which to select values. By default, the flattened input array is used. Default: `None` .
- **mode** (*'raise', 'wrap', 'clip', optional*) –Specifies how out-of-bounds indices will behave. Default: `'clip'` .

  'raise' - raise an error;

  'wrap' - wrap around;

  'clip' - clip to the range. 'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns

Tensor, the indexed result.

Raises

- **ValueError** –If axis is out of range.
- **TypeError** –If the input is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([4, 3, 5, 7, 6, 8])
>>> indices = np.array([0, 1, 4])
>>> output = np.take(a, indices)
>>> print(output)
[4 3 6]
>>> indices = np.array([[0, 1], [2, 3]])
>>> output = np.take(a, indices)
>>> print(output)
[[4 3]
[5 7]]
```

13.2.40 mindspore.numpy.take_along_axis

mindspore.numpy.**take_along_axis**(*arr*, *indices*, *axis*)

Takes values from the input array by matching 1d index and data slices.

This iterates over matching 1d slices oriented along the specified axis in the index and data arrays, and uses the former to look up values in the latter. These slices can be different lengths.

Parameters

- **arr** (Tensor) –Source array with shape *(Ni…, M, Nk…)*.
- **indices** (Tensor) –Indices with shape *(Ni…, J, Nk…)* to take along each 1d slice of *arr*. This must match the dimension of *arr*, but dimensions *Ni* and *Nj* only need to broadcast against *arr*.
- **axis** (*int*) –The axis to take 1d slices along. If *axis* is None, the input array is treated as if it had first been flattened to 1d.

Returns

Tensor, the indexed result, with shape *(Ni…, J, Nk…)*.

Raises

- **ValueError** –If input array and indices have different number of dimensions.
- **TypeError** –If the input is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(12).reshape(3, 4)
>>> indices = np.arange(3).reshape(1, 3)
>>> output = np.take_along_axis(x, indices, 1)
>>> print(output)
[[ 0  1  2]
 [ 4  5  6]
 [ 8  9 10]]
```

13.2.41 mindspore.numpy.tile

mindspore.numpy.**tile**(*a*, *reps*)

Constructs an array by repeating *a* the number of times given by *reps*.

If *reps* has length *d*, the result will have dimension of `max(d, a.ndim)`. If `a.ndim < d`, *a* is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication. If this is not the desired behavior, promote *a* to d-dimensions manually before calling this function. If `a.ndim > d`, *reps* is promoted to `a.ndim` by pre-pending 1's to it. Thus for an *a* of shape (2, 3, 4, 5), a *reps* of (2, 2) is treated as (1, 1, 2, 2).

Parameters

- **a** (Tensor) – The input array.
- **reps** (*int, sequence of ints*) – The number of repetitions of *a* along each axis.

Returns

Tensor, the tiled output array.

Raises

TypeError – If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([0, 1, 2])
>>> output = np.tile(a, 2)
>>> print(output)
[0 1 2 0 1 2]
>>> output = np.tile(a, (2, 2))
>>> print(output)
[[0 1 2 0 1 2]
[0 1 2 0 1 2]]
>>> output = np.tile(a, (2, 1, 2))
>>> print(output)
[[[0 1 2 0 1 2]]
[[0 1 2 0 1 2]]]
```

13.2.42 mindspore.numpy.transpose

mindspore.numpy.**transpose**(*a*, *axes=None*)

Reverses or permutes the axes of a tensor; returns the modified tensor.

Parameters

- **a** (Tensor) – a tensor to be transposed
- **axes** (*Union[None, tuple, list], optional*) – the axes order, if *axes* is *None*, transpose the entire tensor. Default: `None` .

Returns

Tensor, the transposed tensor array.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If the number of *axes* is not equal to a.ndim.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.ones((1,2,3))
>>> x = np.transpose(x)
>>> print(x.shape)
(3, 2, 1)
```

13.2.43 mindspore.numpy.unique

`mindspore.numpy.`**unique**(*x*, *return_inverse=False*)

Finds the unique elements of a tensor. The input tensor will be flattened first when it has more than one dimension.

Note: Numpy arguments *axis*, *return_index* and *return_counts* are not supported. On CPU, this operator must be executed in graph mode.

Parameters

- **x** (`Tensor`) –The input tensor to be processed.
- **return_inverse** (*bool, optional*) –If *True*, also return the indices of the unique tensor. Default: `False` .

Returns

Tensor or tuple of Tensors. If *return_inverse* is *False*, return the unique tensor, otherwise return tuple of tensors.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

TypeError –If *x* is not tensor.

Examples

```
>>> import mindspore.numpy as np
>>> import mindspore as ms
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> input_x = np.asarray([1, 2, 2, 2, 3, 4, 5]).astype('int32')
>>> output_x = np.unique(input_x)
>>> print(output_x)
[1 2 3 4 5]
```

(continues on next page)

(continued from previous page)

```
>>> output_x = np.unique(input_x, return_inverse=True)
>>> print(output_x)
(Tensor(shape=[5], dtype=Int32, value= [ 1, 2, 3, 4, 5]), Tensor(shape=[7], dtype=Int32,
    value= [0, 1, 1, 1, 2, 3, 4]))
```

13.2.44 mindspore.numpy.unravel_index

mindspore.numpy.**unravel_index**(*indices*, *shape*, *order='C'*)

Converts a flat index or array of flat indices into a tuple of coordinate arrays.

Note: Out-of-bound indices are clipped by the boundaries of *shape* instead of raising an error.

Parameters

- **indices** (*Union[int, float, bool, list, tuple, Tensor]*) – An integer array whose elements are indices into the flattened version of an array of dimensions shape.
- **shape** (*tuple(int)*) – The shape of the array to use for unraveling indices.
- **order** (*Union['C', 'F'], optional*) – Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order. Default: `'C'` .

Returns

Tensor, each array in the tuple has the same shape as the indices array.

Supported Platforms:

`Ascend` `GPU` `CPU`

Raises

ValueError – If *order* is not 'C' or 'F'.

Examples

```
>>> import mindspore.numpy as np
>>> print(np.unravel_index([22, 41, 37], (7,6)))
(Tensor(shape=[3], dtype=Int32, value= [3, 6, 6]),
Tensor(shape=[3], dtype=Int32, value= [4, 5, 1]))
>>> print(np.unravel_index([31, 41, 13], (7,6), order='F'))
(Tensor(shape=[3], dtype=Int32, value= [3, 6, 6]),
Tensor(shape=[3], dtype=Int32, value= [4, 5, 1]))
```

13.2.45 mindspore.numpy.vsplit

mindspore.numpy.**vsplit**(*x*, *indices_or_sections*)

Splits a tensor into multiple sub-tensors vertically (row-wise). It is equivalent to split with $axis = 0$ (default), the array is always split along the first axis regardless of the array dimension.

Parameters

- **x** (Tensor) – A Tensor to be divided.
- **indices_or_sections** (*Union[int, tuple(int), list(int)]*) – If integer, *N*, the tensor will be divided into *N* equal tensors along axis. If tuple(int), list(int) or of sorted integers, the entries indicate where along axis the array is split. For example, $[2, 3]$ would, for $axis = 0$, result in three sub-tensors $x[: 2]$, $x[2 : 3]$ and $x[3 :]$. If an index exceeds the dimension of the array along axis, an empty sub-array is returned correspondingly.

Returns

A list of sub-tensors.

Raises

TypeError – If argument *indices_or_sections* is not integer.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.arange(9).reshape((3, 3)).astype('float32')
>>> output = np.vsplit(input_x, 3)
>>> print(output)
(Tensor(shape=[1, 3], dtype=Float32,
  value=[[ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00]]),
 Tensor(shape=[1, 3], dtype=Float32,
  value=[[ 3.00000000e+00,  4.00000000e+00,  5.00000000e+00]]),
 Tensor(shape=[1, 3], dtype=Float32,
  value=[[ 6.00000000e+00,  7.00000000e+00,  8.00000000e+00]]))
```

13.2.46 mindspore.numpy.vstack

mindspore.numpy.**vstack**(*tup*)

Stacks tensors in sequence vertically. This is equivalent to concatenation along the first axis. 1-D tensors should firstly be reshaped to *(1, N)*, and then be concatenated along the first axis.

Parameters

tup (*Union[Tensor, tuple, list]*) – A sequence of 1-D or 2-D tensors. The tensors must have the same shape along all but the first axis. 1-D tensors must have the same shape.

Returns

Stacked Tensor, formed by stacking the given tensors.

Supported Platforms:

Ascend GPU CPU

Raises

- **TypeError** –If *tup* is not Tensor, list or tuple.
- **ValueError** –If *tup* is empty.

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([1, 2, 3]).astype('int32')
>>> x2 = np.array([4, 5, 6]).astype('int32')
>>> output = np.vstack((x1, x2))
>>> print(output)
[[1 2 3]
 [4 5 6]]
```

13.2.47 mindspore.numpy.where

mindspore.numpy.**where**(*condition*, *x=None*, *y=None*)

Returns elements chosen from *x* or *y* depending on *condition*.

Note: As nonzero is not supported, both *x* and *y* must be provided Tensor input.

Parameters

- **condition** (Tensor) –where True, yield *x*, otherwise yield *y*.
- **x** (Tensor, *optional*) –Values from which to choose. Default: None .
- **y** (Tensor, *optional*) –Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape. Default: None .

Returns

Tensor or scalar, with elements from *x* where *condition* is True, and elements from *y* elsewhere.

Raises

ValueError –If operands cannot be broadcast.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> condition = np.full((1, 1, 2), [False, True])
>>> x = np.full((1, 3, 2), 5)
>>> y = np.full((2, 1, 1), 7)
>>> output = np.where(condition, x, y)
>>> print(output)
[[[7 5]
[7 5]
```

(continues on next page)

(continued from previous page)

```
[7 5]]
[[7 5]
[7 5]
[7 5]]]
```

13.3 Logic

Logic operations define computations related with boolean types. Examples of *equal* and *less* operations are as follows:

```
input_x = np.arange(0, 5)
input_y = np.arange(0, 10, 2)
output = np.equal(input_x, input_y)
print("output of equal:", output)
output = np.less(input_x, input_y)
print("output of less:", output)
```

The result is as follows:

```
output of equal: [ True False False False False]
output of less: [False  True  True  True  True]
```

API Name	Description	Supported Platforms
mindspore.numpy.array_equal	Returns *True* if input arrays have same shapes and all elements equal.	GPU CPU Ascend
mindspore.numpy.array_equiv	Returns *True* if input arrays are shape consistent and all elements equal.	Ascend GPU CPU
mindspore.numpy.equal	Returns the truth value of `(x1 == x2)` element-wise.	Ascend GPU CPU
mindspore.numpy.greater	Returns the truth value of `(x1 > x2)` element-wise.	Ascend GPU CPU
mindspore.numpy.greater_equal	Returns the truth value of `(x1 >= x2)` element-wise.	Ascend GPU CPU
mindspore.numpy.in1d	Tests whether each element of a 1-D array is also present in a second array.	Ascend GPU CPU
mindspore.numpy.isclose	Returns a boolean tensor where two tensors are element-wise equal within a tolerance.	Ascend GPU CPU
mindspore.numpy.isfinite	Tests element-wise for finiteness (not infinity or not Not a Number).	Ascend GPU CPU
mindspore.numpy.isin	Calculates element in *test_elements*, broadcasting over *element* only.	Ascend GPU CPU
mindspore.numpy.isinf	Tests element-wise for positive or negative infinity.	GPU CPU
mindspore.numpy.isnan	Tests element-wise for NaN and return result as a boolean array.	GPU CPU
mindspore.numpy.isneginf	Tests element-wise for negative infinity, returns result as bool array.	GPU CPU
mindspore.numpy.isposinf	Tests element-wise for positive infinity, returns result as bool array.	GPU CPU
mindspore.numpy.isscalar	Returns True if the type of element is a scalar type.	Ascend GPU CPU
mindspore.numpy.less	Returns the truth value of `(x1 < x2)` element-wise.	Ascend GPU CPU

continues on next page

Table 3 – continued from previous page

mindspore.numpy.less_equal	Returns the truth value of `(x1 <= x2)` element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.logical_and	Computes the truth value of *x1* AND *x2* element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.logical_not	Computes the truth value of NOT *a* element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.logical_or	Computes the truth value of *x1* OR *x2* element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.logical_xor	Computes the truth value of *x1* XOR *x2*, element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.not_equal	Returns (x1 != x2) element-wise.	`Ascend` `GPU` `CPU`
mindspore.numpy.signbit	Returns element-wise True where signbit is set (less than zero).	`Ascend` `GPU` `CPU`
mindspore.numpy.sometrue	Tests whether any array element along a given axis evaluates to True.	`Ascend` `GPU` `CPU`

13.3.1 mindspore.numpy.array_equal

`mindspore.numpy.`**`array_equal`**(*a1*, *a2*, *equal_nan=False*)

Returns *True* if input arrays have same shapes and all elements equal.

Note: In mindspore, a bool tensor is returned instead, since in Graph mode, the value cannot be traced and computed at compile time.

Since on Ascend, `nan` is treated differently, currently the argument `equal_nan` is not supported on Ascend.

Parameters

- **a1/a2** (*Union[int, float, bool, list, tuple, Tensor]*) – Input arrays.
- **equal_nan** (*bool, optional*) – Whether to compare NaN's as equal. Default: `False` .

Returns

Scalar bool tensor, value is *True* if inputs are equal, *False* otherwise.

Raises

TypeError – If inputs have types not specified above.

Supported Platforms:

`GPU` `CPU` `Ascend`

Examples

```
>>> import mindspore.numpy as np
>>> a = [0,1,2]
>>> b = [[0,1,2], [0,1,2]]
>>> print(np.array_equal(a,b))
False
```

13.3.2 mindspore.numpy.array_equiv

mindspore.numpy.**array_equiv**(*a1*, *a2*)

Returns *True* if input arrays are shape consistent and all elements equal.

Shape consistent means they are either the same shape, or one input array can be broadcasted to create the same shape as the other one.

Note: In mindspore, a bool tensor is returned instead, since in Graph mode, the value cannot be traced and computed at compile time.

Parameters

a1/a2 (*Union[int, float, bool, list, tuple, Tensor]*) –Input arrays.

Returns

Scalar bool tensor, value is *True* if inputs are equivalent, *False* otherwise.

Raises

TypeError –If inputs have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = [0,1,2]
>>> b = [[0,1,2], [0,1,2]]
>>> print(np.array_equiv(a,b))
True
```

13.3.3 mindspore.numpy.equal

mindspore.numpy.**equal**(*x1*, *x2*, *dtype=None*)

Returns the truth value of `(x1 == x2)` element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –Input array.
- **x2** (`Tensor`) –Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.equal(np.array([0, 1, 3]), np.arange(3))
>>> print(output)
[ True  True False]
```

13.3.4 mindspore.numpy.greater

`mindspore.numpy.`**`greater`**(*x1*, *x2*, *dtype=None*)

Returns the truth value of `(x1 > x2)` element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –Input array.
- **x2** (`Tensor`) –Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*`mindspore.dtype`*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.greater(np.array([4, 2]), np.array([2, 2]))
>>> print(output)
[ True False]
```

13.3.5 mindspore.numpy.greater_equal

mindspore.numpy.**greater_equal**(*x1*, *x2*, *dtype=None*)

Returns the truth value of `(x1 >= x2)` element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) −Input array.
- **x2** (Tensor) −Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) −Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.greater_equal(np.array([4, 2, 1]), np.array([2, 2, 2]))
>>> print(output)
[ True  True False]
```

13.3.6 mindspore.numpy.in1d

mindspore.numpy.**in1d**(*ar1*, *ar2*, *invert=False*)

Tests whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as *ar1* that is True where an element of *ar1* is in *ar2* and False otherwise.

Note: Numpy argument *assume_unique* is not supported since the implementation does not rely on the uniqueness of the input arrays.

Parameters

- **ar1** (*Union[int, float, bool, list, tuple, Tensor]*) −Input array with shape *(M,)*.
- **ar2** (*Union[int, float, bool, list, tuple, Tensor]*) −The values against which to test each value of *ar1*.
- **invert** (*boolean, optional*) −If True, the values in the returned array are inverted (that is, False where an element of *ar1* is in *ar2* and True otherwise). Default: `False` .

Returns

Tensor, with shape *(M,)*. The values `ar1[in1d]` are in *ar2*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
>>> print(mask)
[ True False  True False  True]
>>> mask = np.in1d(test, states, invert=True)
>>> print(mask)
[False  True False  True False]
```

13.3.7 mindspore.numpy.isclose

`mindspore.numpy.`**`isclose`**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a boolean tensor where two tensors are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference ($rtol * abs(b)$) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

Note: For finite values, isclose uses the following equation to test whether two floating point values are equivalent. $absolute(a - b)$ <= $(atol + rtol * absolute(b))$ On Ascend, input arrays containing inf or NaN are not supported.

Parameters

- **a** (*Union[Tensor, list, tuple]*) – Input first tensor to compare.
- **b** (*Union[Tensor, list, tuple]*) – Input second tensor to compare.
- **rtol** (*numbers.Number, optional*) – The relative tolerance parameter (see Note). Default: `1e-05` .
- **atol** (*numbers.Number, optional*) – The absolute tolerance parameter (see Note). Default: `1e-08` .
- **equal_nan** (*bool, optional*) – Whether to compare `NaN` as equal. If True, `NaN` in *a* will be considered equal to `NaN` in *b* in the output tensor. Default: `False` .

Returns

A `bool` tensor of where *a* and *b* are equal within the given tolerance.

Raises

`TypeError` – If inputs have types not specified above.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([0,1,2,float('inf'),float('inf'),float('nan')])
>>> b = np.array([0,1,-2,float('-inf'),float('inf'),float('nan')])
>>> print(np.isclose(a, b))
[ True  True False False  True False]
>>> print(np.isclose(a, b, equal_nan=True))
[ True  True False False  True  True]
```

13.3.8 mindspore.numpy.isfinite

mindspore.numpy.**isfinite**(*x*, *dtype=None*)

Tests element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean array.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **x** (Tensor) – Input values.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, true where *x* is not positive infinity, negative infinity, or NaN; false otherwise. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isfinite(np.array([np.inf, 1., np.nan]).astype('float32'))
>>> print(output)
[False  True False]
```

13.3.9 mindspore.numpy.isin

mindspore.numpy.**isin**(*element*, *test_elements*, *invert=False*)

Calculates element in *test_elements*, broadcasting over *element* only. Returns a boolean array of the same shape as *element* that is True where an element of *element* is in *test_elements* and False otherwise.

Note: Numpy argument *assume_unique* is not supported since the implementation does not rely on the uniqueness of the input arrays.

Parameters

- **element** (*Union[int, float, bool, list, tuple, Tensor]*) –Input array.
- **test_elements** (*Union[int, float, bool, list, tuple, Tensor]*) –The values against which to test each value of *element*.
- **invert** (*boolean, optional*) –If True, the values in the returned array are inverted, as if calculating *element* not in *test_elements*. Default: `False` .

Returns

Tensor, has the same shape as *element*. The values `element[isin]` are in *test_elements*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> element = 2*np.arange(4).reshape((2, 2))
>>> test_elements = [1, 2, 4, 8]
>>> mask = np.isin(element, test_elements)
>>> print(mask)
[[False  True]
 [ True False]]
>>> mask = np.isin(element, test_elements, invert=True)
>>> print(mask)
[[ True False]
 [False  True]]
```

13.3.10 mindspore.numpy.isinf

`mindspore.numpy.`**`isinf`**(*x*, *dtype=None*)

Tests element-wise for positive or negative infinity.

Returns a boolean array of the same shape as *x*, True where `x == +/-inf`, otherwise False.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Only np.float32 is currently supported.

Parameters

- **x** (`Tensor`) –Input values.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, true where *x* is positive or negative infinity, false otherwise. This is a scalar if *x* is a scalar.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isinf(np.array(np.inf, np.float32))
>>> print(output)
True
>>> output = np.isinf(np.array([np.inf, -np.inf, 1.0, np.nan], np.float32))
>>> print(output)
[ True  True False False]
```

13.3.11 mindspore.numpy.isnan

mindspore.numpy.**isnan**(*x*, *dtype=None*)

Tests element-wise for NaN and return result as a boolean array.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Only np.float32 is currently supported.

Parameters

- **x** (Tensor) –Input values.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, true where *x* is NaN, false otherwise. This is a scalar if *x* is a scalar.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isnan(np.array(np.nan, np.float32))
>>> print(output)
True
>>> output = np.isnan(np.array(np.inf, np.float32))
>>> print(output)
False
```

13.3.12 mindspore.numpy.isneginf

mindspore.numpy.**isneginf**(*x*)

Tests element-wise for negative infinity, returns result as bool array.

Note: Numpy argument *out* is not supported. Only np.float32 is currently supported.

Parameters

x (Tensor) –Input values.

Returns

Tensor or scalar, true where *x* is negative infinity, false otherwise. This is a scalar if *x* is a scalar.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isneginf(np.array([-np.inf, 0., np.inf, np.nan], np.float32))
>>> print(output)
[ True False False False]
```

13.3.13 mindspore.numpy.isposinf

mindspore.numpy.**isposinf**(*x*)

Tests element-wise for positive infinity, returns result as bool array.

Note: Numpy argument *out* is not supported. Only np.float32 is currently supported.

Parameters

x (Tensor) –Input values.

Returns

Tensor or scalar, true where *x* is positive infinity, false otherwise. This is a scalar if *x* is a scalar.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isposinf(np.array([-np.inf, 0., np.inf, np.nan], np.float32))
>>> print(output)
[False False  True False]
```

13.3.14 mindspore.numpy.isscalar

mindspore.numpy.**isscalar**(*element*)

Returns True if the type of element is a scalar type.

Note: Only object types recognized by the mindspore parser are supported, which includes objects, types, methods and functions defined within the scope of mindspore. Other built-in types are not supported.

Parameters

element (*any*) –Input argument, can be of any type and shape.

Returns

Boolean, True if *element* is a scalar type, False if it is not.

Raises

TypeError –If the type of *element* is not supported by mindspore parser.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.isscalar(3.1)
>>> print(output)
True
>>> output = np.isscalar(np.array(3.1))
>>> print(output)
False
>>> output = np.isscalar(False)
>>> print(output)
True
>>> output = np.isscalar('numpy')
>>> print(output)
True
```

13.3.15 mindspore.numpy.less

mindspore.numpy.**less**(*x1*, *x2*, *dtype=None*)

Returns the truth value of `(x1 < x2)` element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –input array.
- **x2** (Tensor) –Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.less(np.array([1, 2]), np.array([2, 2]))
>>> print(output)
[ True False]
```

13.3.16 mindspore.numpy.less_equal

`mindspore.numpy.`**`less_equal`**(*x1*, *x2*, *dtype=None*)

Returns the truth value of `(x1 <= x2)` element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –Input array.
- **x2** (`Tensor`) –Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.less_equal(np.array([4, 2, 1]), np.array([2, 2, 2]))
>>> print(output)
[False  True  True]
```

13.3.17 mindspore.numpy.logical_and

mindspore.numpy.**logical_and**(*x1*, *x2*, *dtype=None*)

Computes the truth value of *x1* AND *x2* element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – Input tensor.
- **x2** (Tensor) – Input tensor. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. Boolean result of the logical AND operation applied to the elements of *x1* and *x2*; the shape is determined by broadcasting. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([True, False])
>>> x2 = np.array([False, False])
>>> output = np.logical_and(x1, x2)
>>> print(output)
[False False]
```

13.3.18 mindspore.numpy.logical_not

mindspore.numpy.**logical_not**(*a*, *dtype=None*)

Computes the truth value of NOT *a* element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **a** (Tensor) – The input tensor whose dtype is bool.
- **dtype** (*mindspore.dtype*, optional) – Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. Boolean result with the same shape as *a* of the NOT operation on elements of *a*. This is a scalar if *a* is a scalar.

Raises

TypeError – If the input is not a tensor or its dtype is not bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([True, False])
>>> output = np.logical_not(a)
>>> print(output)
[False  True]
```

13.3.19 mindspore.numpy.logical_or

`mindspore.numpy.`**`logical_or`**(*x1*, *x2*, *dtype=None*)

Computes the truth value of *x1* OR *x2* element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) – Input tensor.
- **x2** (`Tensor`) – Input tensor. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, logical OR operation of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([True, False])
>>> x2 = np.array([False, True])
>>> output = np.logical_or(x1, x2)
>>> print(output)
[ True  True]
```

13.3.20 mindspore.numpy.logical_xor

mindspore.numpy.**logical_xor**(*x1*, *x2*, *dtype=None*)

Computes the truth value of *x1* XOR *x2*, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –Input tensor.
- **x2** (Tensor) –Input tensor. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. Boolean result of the logical XOR operation applied to the elements of *x1* and *x2*; the shape is determined by broadcasting. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([True, False])
>>> x2 = np.array([False, False])
>>> output = np.logical_xor(x1, x2)
>>> print(output)
[True False]
```

13.3.21 mindspore.numpy.not_equal

mindspore.numpy.**not_equal**(*x1*, *x2*, *dtype=None*)

Returns (x1 != x2) element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –First input tensor to be compared.
- **x2** (Tensor) –Second input tensor to be compared.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise comparison of *x1* and *x2*. Typically of type bool, unless *dtype* is passed. This is a scalar if both *x1* and *x2* are scalars.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.asarray([1, 2])
>>> b = np.asarray([[1, 3],[1, 4]])
>>> print(np.not_equal(a, b))
[[False  True]
[False  True]]
```

13.3.22 mindspore.numpy.signbit

mindspore.numpy.**signbit**(*x*, *dtype=None*)

Returns element-wise True where signbit is set (less than zero).

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (*Union[int, float, bool, list, tuple, Tensor]*) –The input value(s).
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor.

Raises

TypeError –If input is not array_like or *dtype* is not *None* or *bool*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1, -2.3, 2.1]).astype('float32')
>>> output = np.signbit(x)
>>> print(output)
[False  True False]
```

13.3.23 mindspore.numpy.sometrue

mindspore.numpy.**sometrue**(*a*, *axis=None*, *keepdims=False*)

Tests whether any array element along a given axis evaluates to True.

Returns single boolean unless axis is not None

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) –Input tensor or object that can be converted to an array.
- **axis** (*Union[None, int, tuple(int)]*) –Axis or axes along which a logical OR reduction is performed. Default: `None` . If None, perform a logical OR over all the dimensions of the input array. If negative, it counts from the last to the first axis. If tuple of integers, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.
- **keepdims** (*bool*) –Default: `False` . If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. If the default value is passed, then keepdims will not be passed through to the any method of sub-classes of ndarray, however any non-default value will be. If the sub-class method does not implement keepdims any exceptions will be raised.

Returns

Returns single boolean unless axis is not None

Raises

- **TypeError** –If input is not array_like or *axis* is not int or tuple of integers or *keepdims* is not bool.
- **ValueError** –If any axis is out of range or duplicate axes exist.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1, -2.3, 2.1]).astype('float32')
>>> output = np.sometrue(x)
>>> print(output)
True
```

13.4 Math

Math operations include basic and advanced math operations on tensors, and they have full support on Numpy broadcasting rules. Here are some examples:

- Sum two tensors

  The following code implements the operation of adding two tensors of *input_x* and *input_y*:

```
input_x = np.full((3, 2), [1, 2])
input_y = np.full((3, 2), [3, 4])
output = np.add(input_x, input_y)
print(output)
```

The result is as follows:

```
[[4 6]
 [4 6]
 [4 6]]
```

- Matrix multiplication

  The following code implements the operation of multiplying two matrices *input_x* and *input_y*:

```
input_x = np.arange(2*3).reshape(2, 3).astype('float32')
input_y = np.arange(3*4).reshape(3, 4).astype('float32')
output = np.matmul(input_x, input_y)
print(output)
```

  The result is as follows:

```
[[20. 23. 26. 29.]
 [56. 68. 80. 92.]]
```

- Take the average along a given axis

  The following code implements the operation of averaging all the elements of *input_x*:

```
input_x = np.arange(6).astype('float32')
output = np.mean(input_x)
print(output)
```

  The result is as follows:

```
2.5
```

- Exponential arithmetic

  The following code implements the operation of the natural constant *e* to the power of *input_x*:

```
input_x = np.arange(5).astype('float32')
output = np.exp(input_x)
print(output)
```

  The result is as follows:

```
[ 1.         2.7182817  7.389056  20.085537  54.59815  ]
```

API Name	Description	Supported Platforms
mindspore.numpy.absolute	Calculates the absolute value element-wise.	Ascend GPU CPU
mindspore.numpy.add	Adds arguments element-wise.	Ascend GPU CPU
mindspore.numpy.amax	Returns the maximum of an array or maximum along an axis.	Ascend GPU CPU
mindspore.numpy.amin	Returns the minimum of an array or minimum along an axis.	Ascend GPU CPU
mindspore.numpy.arccos	Trigonometric inverse cosine, element-wise.	Ascend GPU CPU
mindspore.numpy.arccosh	Inverse hyperbolic cosine, element-wise.	Ascend GPU CPU
mindspore.numpy.arcsin	Inverse sine, element-wise.	Ascend GPU CPU
mindspore.numpy.arcsinh	Inverse hyperbolic sine element-wise.	Ascend GPU CPU
mindspore.numpy.arctan	Trigonometric inverse tangent, element-wise.	Ascend GPU CPU

continues on next page

Table 4 – continued from previous page

mindspore.numpy.arctan2	Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.	Ascend GPU CPU
mindspore.numpy.arctanh	Inverse hyperbolic tangent element-wise.	Ascend CPU
mindspore.numpy.argmax	Returns the indices of the maximum values along an axis.	Ascend GPU CPU
mindspore.numpy.argmin	Returns the indices of the minimum values along an axis.	Ascend GPU CPU
mindspore.numpy.around	Evenly round to the given number of decimals.	Ascend GPU CPU
mindspore.numpy.average	Computes the weighted average along the specified axis.	Ascend GPU CPU
mindspore.numpy.bincount	Count number of occurrences of each value in array of non-negative ints.	Ascend GPU CPU
mindspore.numpy.bitwise_and	Computes the bit-wise AND of two arrays element-wise.	Ascend CPU
mindspore.numpy.bitwise_or	Computes the bit-wise OR of two arrays element-wise.	Ascend CPU
mindspore.numpy.bitwise_xor	Computes the bit-wise XOR of two arrays element-wise.	Ascend CPU
mindspore.numpy.cbrt	Returns the cube-root of a tensor, element-wise.	Ascend GPU CPU
mindspore.numpy.ceil	Returns the ceiling of the input, element-wise.	Ascend GPU CPU
mindspore.numpy.clip	Clips (limits) the values in an array.	Ascend GPU CPU
mindspore.numpy.convolve	Returns the discrete, linear convolution of two one-dimensional sequences.	GPU CPU
mindspore.numpy.copysign	Changes the sign of *x1* to that of *x2*, element-wise.	Ascend GPU CPU
mindspore.numpy.corrcoef	Returns Pearson product-moment correlation coefficients.	Ascend GPU CPU
mindspore.numpy.correlate	Cross-correlation of two 1-dimensional sequences.	Ascend GPU CPU
mindspore.numpy.cos	Cosine element-wise.	Ascend GPU CPU
mindspore.numpy.cosh	Hyperbolic cosine, element-wise.	Ascend CPU
mindspore.numpy.count_nonzero	Counts the number of non-zero values in the tensor *x*.	Ascend GPU CPU
mindspore.numpy.cov	Estimates a covariance matrix, given data and weights.	Ascend GPU CPU
mindspore.numpy.cross	Returns the cross product of two (arrays of) vectors.	Ascend GPU CPU
mindspore.numpy.cumprod	Returns the cumulative product of elements along a given axis.	Ascend GPU
mindspore.numpy.cumsum	Returns the cumulative sum of the elements along a given axis.	Ascend GPU CPU
mindspore.numpy.deg2rad	Converts angles from degrees to radians.	Ascend GPU CPU
mindspore.numpy.diff	Calculates the n-th discrete difference along the given axis.	Ascend GPU CPU
mindspore.numpy.digitize	Returns the indices of the bins to which each value in input array belongs.	Ascend GPU CPU
mindspore.numpy.divide	Returns a true division of the inputs, element-wise.	Ascend GPU CPU
mindspore.numpy.divmod	Returns element-wise quotient and remainder simultaneously.	Ascend GPU CPU
mindspore.numpy.dot	Returns the dot product of two arrays.	Ascend GPU CPU
mindspore.numpy.ediff1d	The differences between consecutive elements of a tensor.	Ascend GPU CPU
mindspore.numpy.exp	Calculates the exponential of all elements in the input array.	Ascend GPU CPU

continues on next page

Table 4 – continued from previous page

mindspore.numpy.exp2	Calculates 2**p for all p in the input array.	Ascend GPU CPU
mindspore.numpy.expm1	Calculates exp(x) - 1 for all elements in the array.	Ascend GPU CPU
mindspore.numpy.fix	Rounds to nearest integer towards zero.	Ascend GPU CPU
mindspore.numpy.float_power	First array elements raised to powers from second array, element-wise.	Ascend GPU CPU
mindspore.numpy.floor	Returns the floor of the input, element-wise.	Ascend GPU CPU
mindspore.numpy.floor_divide	Returns the largest integer smaller or equal to the division of the inputs.	Ascend GPU CPU
mindspore.numpy.fmod	Returns the element-wise remainder of division.	Ascend GPU CPU
mindspore.numpy.gcd	Returns the greatest common divisor of |x1| and |x2|.	Ascend GPU CPU
mindspore.numpy.gradient	Returns the gradient of a N-dimensional array.	Ascend GPU CPU
mindspore.numpy.heaviside	Computes the Heaviside step function.	Ascend GPU CPU
mindspore.numpy.histogram	Computes the histogram of a dataset.	Ascend GPU CPU
mindspore.numpy.histogram2d	Computes the multidimensional histogram of some data.	Ascend GPU CPU
mindspore.numpy.histogramdd	Computes the multidimensional histogram of some data.	Ascend GPU CPU
mindspore.numpy.hypot	Given the "legs" of a right triangle, returns its hypotenuse.	Ascend GPU CPU
mindspore.numpy.inner	Returns the inner product of two tensors.	Ascend GPU CPU
mindspore.numpy.interp	One-dimensional linear interpolation for monotonically increasing sample points.	Ascend GPU CPU
mindspore.numpy.invert	Computes bit-wise inversion, or bit-wise NOT, element-wise.	Ascend
mindspore.numpy.kron	Kronecker product of two arrays.	Ascend GPU CPU
mindspore.numpy.lcm	Returns the lowest common multiple of |x1| and |x2|.	Ascend GPU CPU
mindspore.numpy.log	Returns the natural logarithm, element-wise.	Ascend GPU CPU
mindspore.numpy.log10	Base-10 logarithm of *x*.	Ascend GPU CPU
mindspore.numpy.log1p	Returns the natural logarithm of one plus the input array, element-wise.	Ascend GPU CPU
mindspore.numpy.log2	Base-2 logarithm of *x*.	Ascend GPU CPU
mindspore.numpy.logaddexp	Logarithm of the sum of exponentiations of the inputs.	Ascend GPU CPU
mindspore.numpy.logaddexp2	Logarithm of the sum of exponentiations of the inputs in base of 2.	Ascend GPU CPU
mindspore.numpy.matmul	Returns the matrix product of two arrays.	Ascend GPU CPU
mindspore.numpy.matrix_power	Raises a square matrix to the (integer) power *n*.	Ascend GPU CPU
mindspore.numpy.maximum	Returns the element-wise maximum of array elements.	Ascend GPU CPU
mindspore.numpy.mean	Computes the arithmetic mean along the specified axis.	Ascend GPU CPU
mindspore.numpy.minimum	Element-wise minimum of tensor elements.	Ascend GPU CPU
mindspore.numpy.multi_dot	Computes the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.	Ascend GPU CPU
mindspore.numpy.multiply	Multiplies arguments element-wise.	Ascend GPU CPU
mindspore.numpy.nancumsum	Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.	GPU CPU

continues on next page

Table 4 – continued from previous page

mindspore.numpy.nanmax	Return the maximum of an array or maximum along an axis, ignoring any NaNs.	GPU CPU
mindspore.numpy.nanmean	Computes the arithmetic mean along the specified axis, ignoring NaNs.	GPU CPU
mindspore.numpy.nanmin	Returns the minimum of array elements over a given axis, ignoring any NaNs.	GPU CPU
mindspore.numpy.nanstd	Computes the standard deviation along the specified axis, while ignoring NaNs.	GPU CPU
mindspore.numpy.nansum	Returns the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.	GPU CPU
mindspore.numpy.nanvar	Computes the variance along the specified axis, while ignoring NaNs.	GPU CPU
mindspore.numpy.negative	Numerical negative, element-wise.	Ascend GPU CPU
mindspore.numpy.norm	Matrix or vector norm.	Ascend GPU CPU
mindspore.numpy.outer	Computes the outer product of two vectors.	Ascend GPU CPU
mindspore.numpy.polyadd	Finds the sum of two polynomials.	Ascend GPU CPU
mindspore.numpy.polyder	Returns the derivative of the specified order of a polynomial.	Ascend GPU CPU
mindspore.numpy.polyint	Returns an antiderivative (indefinite integral) of a polynomial.	Ascend GPU CPU
mindspore.numpy.polymul	Finds the product of two polynomials.	GPU
mindspore.numpy.polysub	Difference (subtraction) of two polynomials.	Ascend GPU CPU
mindspore.numpy.polyval	Evaluates a polynomial at specific values.	Ascend GPU CPU
mindspore.numpy.positive	Numerical positive, element-wise.	Ascend GPU CPU
mindspore.numpy.power	First array elements raised to powers from second array, element-wise.	Ascend GPU CPU
mindspore.numpy.promote_types	Returns the data type with the smallest size and smallest scalar kind.	Ascend GPU CPU
mindspore.numpy.ptp	Range of values (maximum - minimum) along an axis.	Ascend GPU CPU
mindspore.numpy.rad2deg	Converts angles from radians to degrees.	Ascend GPU CPU
mindspore.numpy.radians	Converts angles from degrees to radians.	Ascend GPU CPU
mindspore.numpy.ravel_multi_index	Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.	GPU
mindspore.numpy.reciprocal	Returns the reciprocal of the argument, element-wise.	Ascend GPU CPU
mindspore.numpy.remainder	Returns element-wise remainder of division.	Ascend GPU CPU
mindspore.numpy.result_type	Returns the type that results from applying the type promotion rules to the arguments.	Ascend GPU CPU
mindspore.numpy.rint	Rounds elements of the array to the nearest integer.	Ascend GPU CPU
mindspore.numpy.searchsorted	Finds indices where elements should be inserted to maintain order.	Ascend GPU CPU
mindspore.numpy.sign	Returns an element-wise indication of the sign of a number.	Ascend GPU CPU
mindspore.numpy.sin	Trigonometric sine, element-wise.	Ascend GPU CPU
mindspore.numpy.sinh	Hyperbolic sine, element-wise.	Ascend CPU
mindspore.numpy.sqrt	Returns the non-negative square-root of an array, element-wise.	Ascend GPU CPU
mindspore.numpy.square	Returns the element-wise square of the input.	Ascend GPU CPU

continues on next page

Table 4 – continued from previous page

mindspore.numpy.std	Computes the standard deviation along the specified axis.	Ascend GPU CPU
mindspore.numpy.subtract	Subtracts arguments, element-wise.	Ascend GPU CPU
mindspore.numpy.sum	Returns sum of array elements over a given axis.	Ascend GPU CPU
mindspore.numpy.tan	Computes tangent element-wise.	Ascend CPU
mindspore.numpy.tanh	Computes hyperbolic tangent element-wise.	Ascend GPU CPU
mindspore.numpy.tensordot	Computes tensor dot product along specified axes.	Ascend GPU CPU
mindspore.numpy.trapz	Integrates along the given axis using the composite trapezoidal rule.	Ascend GPU CPU
mindspore.numpy.true_divide	Returns a true division of the inputs, element-wise.	Ascend GPU CPU
mindspore.numpy.trunc	Returns the truncated value of the input, element-wise.	Ascend GPU CPU
mindspore.numpy.unwrap	Unwraps by changing deltas between values to 2*pi complement.	Ascend GPU CPU
mindspore.numpy.var	Computes the variance along the specified axis.	Ascend GPU CPU

13.4.1 mindspore.numpy.absolute

mindspore.numpy.**absolute**(*x*, *dtype=None*)

Calculates the absolute value element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Currently the backend kernel only supports float calculation, if the input is not a *float*, then it will be casted to mstype.float32 and casted back.

Parameters

- **x** (Tensor) –Tensor to be used for calculation.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor.

Raises

TypeError –If input arguments have types not specified above.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, -5], np.float32)
>>> output = np.absolute(x)
>>> print(output)
[1. 2. 3. 4. 5.]
```

13.4.2 mindspore.numpy.add

mindspore.numpy.**add**(*x1*, *x2*, *dtype=None*)

Adds arguments element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input to be added.
- **x2** (Tensor) – input to be added.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the sum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2])
>>> x2 = np.full((3, 2), [3, 4])
>>> output = np.add(x1, x2)
>>> print(output)
[[4 6]
[4 6]
[4 6]]
```

13.4.3 mindspore.numpy.amax

mindspore.numpy.**amax**(*a*, *axis=None*, *keepdims=False*, *initial=None*, *where=True*)

Returns the maximum of an array or maximum along an axis.

Note: Numpy argument *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **a** (Tensor) – Input data.
- **axis** (*Union[int, tuple(int), None], optional*) – Default: None . Axis or axes along which to operate. By default, flattened input is used. If this is a tuple of integers, the maximum is selected over multiple axes, instead of a single axis or all the axes as before.
- **keepdims** (*boolean, optional*) – Default: False . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
- **initial** (*scalar, optional*) – Default: None . The minimum value of an output element. Must be present to allow computation on empty slice.

- **where** (*boolean Tensor, optional*) –Default: `True` . A boolean array which is broadcasted to match the dimensions of array, and selects elements to include in the reduction. If non-default value is passed, initial must also be provided.

Returns

Tensor or scalar, maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(4).reshape((2,2)).astype('float32')
>>> output = np.amax(a)
>>> print(output)
3.0
>>> output = np.amax(a, axis=0)
>>> print(output)
[2. 3.]
>>> output = np.amax(a, axis=1)
>>> print(output)
[1. 3.]
>>> output = np.amax(a, where=np.array([False, True]), initial=-1, axis=0)
>>> print(output)
[-1.  3.]
```

13.4.4 mindspore.numpy.amin

`mindspore.numpy.`**`amin`**(*a*, *axis=None*, *keepdims=False*, *initial=None*, *where=True*)

Returns the minimum of an array or minimum along an axis.

Note: Numpy argument *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **a** (`Tensor`) –Input data.
- **axis** (*`Union[int, tuple(int), None], optional`*) –Default: `None` . Axis or axes along which to operate. By default, flattened input is used. If this is a tuple of integers, the minimum is selected over multiple axes, instead of a single axis or all the axes as before.
- **keepdims** (*`bool, optional`*) –Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
- **initial** (*`Number, optional`*) –Default: `None` . The maximum value of an output element. Must be present to allow computation on empty slice.

- **where** (*bool Tensor, optional*) –Default: `True` . A boolean array which is broadcasted to match the dimensions of array, and selects elements to include in the reduction. If non-default value is passed, initial must also be provided.

Returns

Tensor or scalar, minimum of *a*. If axis is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(4).reshape((2,2)).astype('float32')
>>> output = np.amin(a)
>>> print(output)
0.0
>>> output = np.amin(a, axis=0)
>>> print(output)
[0. 1.]
>>> output = np.amin(a, axis=1)
>>> print(output)
[0. 2.]
>>> output = np.amin(a, where=np.array([False, True]), initial=10, axis=0)
>>> print(output)
[10.  1.]
```

13.4.5 mindspore.numpy.arccos

`mindspore.numpy.`**`arccos`**(*input*, *dtype=None*)

Trigonometric inverse cosine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **input** (`Tensor`) –Input tensor. x-coordinate on the unit circle. For real arguments, the domain is $[-1, 1]$.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input = np.asarray([1, -1], np.float32)
>>> output = np.arccos(input)
>>> print(output)
[0.        3.1415927]
```

13.4.6 mindspore.numpy.arccosh

mindspore.numpy.**arccosh**(*x*, *dtype=None*)

Inverse hyperbolic cosine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(1, 5).astype('float32')
>>> print(np.arccosh(x))
[0.        1.316958  1.7627472 2.063437 ]
```

13.4.7 mindspore.numpy.arcsin

mindspore.numpy.**arcsin**(*x*, *dtype=None*)

Inverse sine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor. y-coordinate on the unit circle.
- **dtype** (*mindspore.dtype*, optional) –Default: None. Overrides the dtype of the output Tensor.

Returns

Output Tensor.

Raises

TypeError –If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, -1], np.float32)
>>> output = np.arcsin(x)
>>> print(output)
[ 1.5707964 -1.5707964]
```

13.4.8 mindspore.numpy.arcsinh

`mindspore.numpy.`**`arcsinh`**(*x*, *dtype=None*)

Inverse hyperbolic sine element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (`Tensor`) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1., 2., 3., 4.], dtype=np.float32)
>>> print(np.arcsinh(x))
[0.8813736 1.4436355 1.8184465 2.0947125]
```

13.4.9 mindspore.numpy.arctan

mindspore.numpy.**arctan**(*x*, *dtype=None*)

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if $y = tan(x)$ then $x = arctan(y)$.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5).astype('float32')
>>> print(np.arctan(x))
[0.        0.7853982 1.1071488 1.2490457 1.3258177]
```

13.4.10 mindspore.numpy.arctan2

mindspore.numpy.**arctan2**(*x1*, *x2*, *dtype=None*)

Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input tensor.
- **x2** (Tensor) – input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the sum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([-1, +1, +1, -1])
>>> x2 = np.array([-1, -1, +1, +1])
>>> output = np.arctan2(x1, x2)
>>> print(output)
[-2.3561945   2.3561945   0.78539819 -0.78539819]
```

13.4.11 mindspore.numpy.arctanh

mindspore.numpy.**arctanh**(*x*, *dtype=None*)

Inverse hyperbolic tangent element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([-0.99, -0.75, -0.5, 0, 0.5]).astype('float32')
>>> print(np.arctanh(x))
[-2.646653   -0.97295505 -0.54930615  0.          0.54930615]
```

13.4.12 mindspore.numpy.argmax

mindspore.numpy.**argmax**(*a*, *axis=None*)

Returns the indices of the maximum values along an axis.

Note: Numpy argument *out* is not supported. On Ascend, in case of multiple occurrences of the maximum values, the return indices may not necessarily correspond to the first occurrence.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input array.
- **axis** (*int, optional*) – By default, the index is into the flattened array, otherwise along the specified axis. Default: None .

Returns

Tensor, array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.

Raises

ValueError – If axis is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(10, 16).reshape(2, 3)
>>> print(np.argmax(a))
5
>>> print(np.argmax(a, axis=0))
[1 1 1]
>>> print(np.argmax(a, axis=1))
[2 2]
```

13.4.13 mindspore.numpy.argmin

`mindspore.numpy.`**`argmin`**(*a*, *axis=None*)

Returns the indices of the minimum values along an axis.

Note: Numpy argument *out* is not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input array.
- **axis** (*int, optional*) – By default, the index is into the flattened array, otherwise along the specified axis. Default: `None` .

Returns

Tensor, array of indices into the array. It has the same shape as a.shape with the dimension along axis removed.

Raises

ValueError – If axis is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(10, 16).reshape(2, 3)
>>> print(np.argmin(a))
0
>>> print(np.argmin(a, axis=0))
[0 0 0]
>>> print(np.argmin(a, axis=1))
[0 0]
```

13.4.14 mindspore.numpy.around

mindspore.numpy.**around**(*a*, *decimals=0*)

Evenly round to the given number of decimals.

Note: Numpy argument *out* is not supported. Complex numbers are not supported.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) – Input data.
- **decimals** (*int*) – Number of decimal places to round to. Default: 0 .

Returns

Tensor. A tensor of the same type as a, containing the rounded values. The result of rounding a float is a float.

Raises

TypeError – If the input can not be converted to a tensor or the *decimals* argument is not integer.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([-1.3, 0.0, 0.5, 1.5, 2.5])
>>> print(np.around(a))
[-1. 0. 0. 2. 2.]
```

13.4.15 mindspore.numpy.average

mindspore.numpy.**average**(*x*, *axis=None*, *weights=None*, *returned=False*)

Computes the weighted average along the specified axis.

Parameters

- **x** (Tensor) – A Tensor to be averaged.

- **axis** (*Union[None, int, tuple(int)], optional*) –Axis along which to average *x*. Default: `None` . If the axis is *None*, it will average over all of the elements of the tensor *x*. If the axis is negative, it counts from the last to the first axis.
- **weights** (*Union[None, Tensor], optional*) –Weights associated with the values in *x*. Default: `None` . If *weights* is *None*, all the data in *x* are assumed to have a weight equal to one. If *weights* is 1-D tensor, the length must be the same as the given axis. Otherwise, *weights* should have the same shape as *x*.
- **returned** (*bool, optional*) –Default: `False` . If *True*, the tuple (average, sum_of_weights) is returned. If *False*, only the average is returned.

Returns

Averaged Tensor. If returned is *True*, return tuple.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.array([[1., 2.], [3., 4.]])
>>> output = np.average(input_x, axis=0, weights=input_x, returned=True)
>>> print(output)
(Tensor(shape=[2], dtype=Float32, value= [ 2.50000000e+00,  3.33333325e+00]),
 Tensor(shape=[2], dtype=Float32, value= [ 4.00000000e+00,  6.00000000e+00]))
```

13.4.16 mindspore.numpy.bincount

`mindspore.numpy.`**bincount**(*x*, *weights=None*, *minlength=0*, *length=None*)

Count number of occurrences of each value in array of non-negative ints. The number of bins (of size 1) is one larger than the largest value in *x*. If *minlength* is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of *x*). Each bin gives the number of occurrences of its index value in *x*. If *weights* is specified the input array is weighted by it, i.e. if a value *n* is found at position *i*, `out[n] += weight[i]` instead of `out[n] += 1`.

Note: The additional argument *length* specifies the number of bins (overriding `x.max() + 1`), which must be provided in graph mode. If *x* contains negative values, no error will be raised, and negative values are treated as zeros instead.

Parameters

- **x** (*Union[list, tuple, Tensor]*) –1-d input array.
- **weights** (*Union[int, float, bool, list, tuple, Tensor], optional*) –Weights, array of the same shape as *x*. Default: `None` .
- **minlength** (*int, optional*) –A minimum number of bins for the output array. Default: `0` .
- **length** (*int, optional*) –Number of bins. Default: `None` .

Returns

Tensor, the result of binning the input array. The length of out is equal to `np.amax(x)+1`.

Raises

ValueError –If *x* is not one-dimensional, or if *x* and *weights* do not have the same shape.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.bincount(np.arange(5)))
[1. 1. 1. 1. 1.]
>>> print(np.bincount(np.array([0, 1, 1, 3, 2, 1, 7])))
[1. 3. 1. 1. 0. 0. 0. 1.]
>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> print(np.bincount(x,  weights=w))
[0.3 0.7 1.1]
```

13.4.17 mindspore.numpy.bitwise_and

`mindspore.numpy.`**`bitwise_and`**(*x1*, *x2*, *dtype=None*)

Computes the bit-wise AND of two arrays element-wise. Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator &.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –Input array.
- **x2** (`Tensor`) –Input array. Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if both x1 and x2 are scalars.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.bitwise_and(13, 17))
1
```

13.4.18 mindspore.numpy.bitwise_or

mindspore.numpy.**bitwise_or**(*x1*, *x2*, *dtype=None*)

Computes the bit-wise OR of two arrays element-wise. Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator |.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –Input array.
- **x2** (Tensor) –Input array. Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if both x1 and x2 are scalars.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.bitwise_or(13, 16))
29
```

13.4.19 mindspore.numpy.bitwise_xor

mindspore.numpy.**bitwise_xor**(*x1*, *x2*, *dtype=None*)

Computes the bit-wise XOR of two arrays element-wise. Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ^.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –Input array.
- **x2** (Tensor) –Input array. Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if both x1 and x2 are scalars.

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.bitwise_xor(13, 17))
28
```

13.4.20 mindspore.numpy.cbrt

mindspore.numpy.**cbrt**(*x*, *dtype=None*)

Returns the cube-root of a tensor, element-wise.

Note: Numpy arguments *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.asarray([1, -1, 3, -8, 64])
>>> output = np.cbrt(a)
>>> print(output)
[ 1.         -1.          1.4422495 -2.          4.        ]
```

13.4.21 mindspore.numpy.ceil

mindspore.numpy.**ceil**(*x*, *dtype=None*)

Returns the ceiling of the input, element-wise.

The ceil of the scalar *x* is the smallest integer *i*, such that i >= x.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **x** (Tensor) – input values.

- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the floor of each element in *x*. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> output = np.ceil(a)
>>> print(output)
[-1. -1. -0.  1.  2.  2.  2.]
```

13.4.22 mindspore.numpy.clip

`mindspore.numpy.clip`(*x*, *xmin*, *xmax*, *dtype=None*)

Clips (limits) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of $[0, 1]$ is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

- **x** (`Tensor`) –Tensor containing elements to clip.
- **xmin** (`Tensor, scalar, None`) –Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of *xmin* and *xmax* may be None.
- **xmax** (`Tensor, scalar, None`) –Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of *xmin* and *xmax* may be None. If *xmin* or *xmax* are tensors, then the three tensors will be broadcasted to match their shapes.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor, a tensor with the elements of *x*, but where values < *xmin* are replaced with *xmin*, and those > *xmax* with *xmax*.

Raises

- **TypeError** –If inputs have types not specified above.
- **ValueError** –If the shapes of *x1* and *x2* cannot broadcast, or both *xmin* and *xmax* are *None*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, 0, 3, 2, 0])
>>> output = np.clip(x, 0, 2)
>>> print(output)
[1 2 2 0 0 2 2 0]
```

13.4.23 mindspore.numpy.convolve

mindspore.numpy.**convolve**(*a*, *v*, *mode='full'*)

Returns the discrete, linear convolution of two one-dimensional sequences.

Note: If *v* is longer than *a*, the tensors are swapped before computation.

Parameters

- **a** (*Union[list, tuple, Tensor]*) – First one-dimensional input tensor.
- **v** (*Union[list, tuple, Tensor]*) – Second one-dimensional input tensor.
- **mode** (*str, optional*) – By default, mode is *'full'*. This returns the convolution at each point of overlap, with an output shape of $(N+M-1,)$. At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen. If *mode* is *'same'*, it returns output of length $max(M, N)$. Boundary effects are still visible. If *mode* is *'valid'*, it returns output of length $max(M, N) - min(M, N) + 1$. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Returns

Tensor, discrete, linear convolution of a and v.

Raises

- **TypeError** – If the inputs have types not specified above.
- **ValueError** – If a and v are empty or have wrong dimensions

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.convolve([1., 2., 3., 4., 5.], [2., 3.], mode="valid")
>>> print(output)
[ 7. 12. 17. 22.]
```

13.4.24 mindspore.numpy.copysign

mindspore.numpy.**copysign**(*x1*, *x2*, *dtype=None*)

Changes the sign of *x1* to that of *x2*, element-wise.

If *x2* is a scalar, its sign will be copied to all elements of *x1*.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Complex inputs are not supported now.

Parameters

- **x1** (*Union[int, float, list, tuple, Tensor]*) – Values to change the sign of.
- **x2** (*Union[int, float, list, tuple, Tensor]*) – The sign of x2 is copied to x1. If *x1.shape != x2.shape*, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. The values of *x1* with the sign of *x2*. This is a scalar if both *x1* and *x2* are scalars.

Raises

TypeError – If dtype of the input is not in the given types or the input can not be converted to tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.copysign(np.array([1, -1, -1]), np.array([-1, 1, -1]))
>>> print(output)
[-1  1 -1]
```

13.4.25 mindspore.numpy.corrcoef

mindspore.numpy.**corrcoef**(*x*, *y=None*, *rowvar=True*, *dtype=None*)

Returns Pearson product-moment correlation coefficients.

Please refer to the documentation for cov for more detail. The relationship between the correlation coefficient matrix, R, and the covariance matrix, C, is $R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$ The values of R are between -1 and 1, inclusive.

Note: Currently, complex numbers are not supported.

Parameters

- **x** (*Union[int, float, bool, tuple, list, Tensor]*) – A 1-D or 2-D array containing multiple variables and observations. Each row of *x* represents a variable, and each column a single observation of all those variables. Also see rowvar below.

- **y** (*Union[int, float, bool, tuple, list, Tensor], optional*) –An additional set of variables and observations. Default: `None` .
- **rowvar** (*bool, optional*) –If rowvar is `True` (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations. Default: `True` .
- **dtype** (*mindspore.dtype*, optional) –Data-type of the result. By default, the return data-type will have at least float32 precision. Default: `None` .

Returns

Tensor, The correlation coefficient matrix of the variables.

Raises

- **TypeError** –If the inputs have types not specified above.
- **ValueError** –If *x* and *y* have wrong dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.corrcoef([[2., 3., 4., 5.], [0., 2., 3., 4.], [7., 8., 9., 10.]])
>>> print(output)
[[1.         0.9827076  1.        ]
[0.9827077  0.99999994 0.9827077 ]
[1.         0.9827076  1.        ]]
```

13.4.26 mindspore.numpy.correlate

`mindspore.numpy.`**correlate**(*a*, *v*, *mode='valid'*)

Cross-correlation of two 1-dimensional sequences.

This function computes the correlation as generally defined in signal processing texts:

$c_{av}[k] = \sum_n a[n+k] * conj(v[n])$

with *a* and *v* sequences being zero-padded where necessary and conj being the conjugate.

Note:

- *correlate* is currently only used in *mindscience* scientific computing scenarios and dose not support other usage scenarios.
- *correlate* is not supported on Windows platform yet.

Parameters

- **a** (*Union[list, tuple, Tensor]*) –First input sequence.
- **v** (*Union[list, tuple, Tensor]*) –Second input sequence.
- **mode** (*str, optional*) –Specifies padding mode. The optional values are `"same"` , `"valid"` and `"full"` . Default: `"valid"` .

- "same": it returns output of length $max(M, N)$. Boundary effects are still visible.
- "valid": it returns output of length $max(M, N) - min(M, N) + 1$. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.
- "full": it returns the convolution at each point of overlap, with an output shape of $(N+M-1,)$.At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.

Returns

Tensor, Discrete cross-correlation of *a* and *v*.

Raises

- **TypeError** –If *a* or *v* is not a tensor.
- **TypeError** –If *a* and *v* is of different dtype.
- **ValueError** –If *a* and *v* are empty or have wrong dimensions

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as mnp
>>> from mindspore import Tensor
>>> output = mnp.correlate(Tensor([1., 2., 3.]), Tensor([0., 1., 0.5]))
>>> print(output)
[3.5]
>>> output = mnp.correlate(Tensor([1., 2., 3.]), Tensor([0., 1., 0.5]), mode="same")
>>> print(output)
[2.  3.5 3. ]
>>> output = mnp.correlate(Tensor([1., 2., 3., 4., 5.]), Tensor([1., 2.]), mode="full")
>>> print(output)
[ 2.  5.  8. 11. 14.  5.]
```

13.4.27 mindspore.numpy.cos

mindspore.numpy.**cos**(*x*, *dtype=None*)

Cosine element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5).astype('float32')
>>> print(np.cos(x))
[ 1.          0.5403023  -0.41614684 -0.9899925  -0.6536436 ]
```

13.4.28 mindspore.numpy.cosh

mindspore.numpy.**cosh**(*x*, *dtype=None*)

Hyperbolic cosine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5).astype('float32')
>>> print(np.cosh(x))
[ 1.          1.5430807   3.7621956  10.067662   27.308233 ]
```

13.4.29 mindspore.numpy.count_nonzero

mindspore.numpy.**count_nonzero**(*x*, *axis=None*, *keepdims=False*)

Counts the number of non-zero values in the tensor *x*.

Parameters

- **x** (Tensor) – The tensor for which to count non-zeros.
- **axis** (*Union[int, tuple], optional*) – Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of *x*. Default: None .
- **keepdims** (*bool, optional*) – If this is set to True, the axes that are counted are left in the result as dimensions with size one. With this option, the result will broadcast correctly against *x*. Default: False .

Returns

Tensor, indicating number of non-zero values in the *x* along a given axis. Otherwise, the total number of non-zero values in *x* is returned.

Raises

- **TypeError** –If axis is not int or tuple.
- **ValueError** –If axis is not in range [-x.ndim, x.ndim).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, 0, 3, 2, 0])
>>> output = np.count_nonzero(x)
>>> print(output)
6
```

13.4.30 mindspore.numpy.cov

`mindspore.numpy.`**cov**(*m*, *y=None*, *rowvar=True*, *bias=False*, *ddof=None*, *fweights=None*, *aweights=None*, *dtype=None*)

Estimates a covariance matrix, given data and weights.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, $X = [x_1, x_2, ..x_N]^T$, then the covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i.

Note: *fweights* and *aweights* must be all positive, in Numpy if negative values are detected, a value error will be raised, in MindSpore we converts all values to positive instead.

Parameters

- **m** (*Union[Tensor, list, tuple]*) –A 1-D or 2-D tensor containing multiple variables and observations. Each row of *m* represents a variable, and each column represents a single observation of all those variables. Also see *rowvar* below.
- **y** (*Union[Tensor, list, tuple], optional*) –An additional set of variables and observations. *y* has the same form as that of *m*, default is `None`.
- **rowvar** (*bool, optional*) –If *rowvar* is `True` (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.
- **bias** (*bool, optional*) –Default Normalization (`False`) is by $(N - 1)$, where N is the number of observations given (unbiased estimate). If bias is `True`, then Normalization is by N. These values can be overridden by using the keyword *ddof*.
- **ddof** (*int, optional*) –If not `None`, the default value implied by *bias* is overridden. Note that $ddof = 1$ will return the unbiased estimate, even if both fweights and aweights are specified, and $ddof = 0$ will return the simple average. See the notes for the details. The default value is `None`.
- **fweights** (*Union[Tensor, list, tuple], optional*) –1-D tensor of integer frequency weights; the number of times each observation vector should be repeated. The default value is `None`.
- **aweights** (*Union[Tensor, list, tuple], optional*) –1-D tensor of observation vector weights. These relative weights are typically larger for observations considered more important and smaller for observations considered

less important. If $ddof = 0$ the tensor of weights can be used to assign probabilities to observation vectors. The default value is `None`.

- **dtype** (Union[*mindspore.dtype*, str], optional) −Data-type of the result. By default, the return data-type will have mstype.float32 precision. Default is `None`.

Returns

Tensor, the covariance matrix of the variables.

Raises

- **TypeError** −If the inputs have types not specified above.
- **ValueError** −If *m* and *y* have wrong dimensions.
- **RuntimeError** −If *aweights* and *fweights* have dimensions > 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.cov([[2., 3., 4., 5.], [0., 2., 3., 4.], [7., 8., 9., 10.]])
>>> print(output)
[[1.6666666 2.1666667 1.6666666]
[2.1666667 2.9166667 2.1666667]
[1.6666666 2.1666667 1.6666666]]
```

13.4.31 mindspore.numpy.cross

`mindspore.numpy.`**cross**(*a*, *b*, *axisa=- 1*, *axisb=- 1*, *axisc=- 1*, *axis=None*)

Returns the cross product of two (arrays of) vectors.

The cross product of a and b in R^3 is a vector perpendicular to both a and b. If a and b are arrays of vectors, the vectors are defined by the last axis of a and b by default, and these axes can have dimensions 2 or 3. Where the dimension of either a or b is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

Parameters

- **a** (*Union[list, tuple, Tensor]*) −Components of the first vector(s).
- **b** (*Union[list, tuple, Tensor]*) −Components of the second vector(s).
- **axisa** (*int, optional*) −Axis of *a* that defines the vector(s). By default, the last axis.
- **axisb** (*int, optional*) −Axis of *b* that defines the vector(s). By default, the last axis.
- **axisc** (*int, optional*) −Axis of *c* containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.
- **axis** (*int, optional*) −If defined, the axis of *a*, *b* and *c* that defines the vector(s) and cross product(s). Overrides *axisa*, *axisb* and *axisc*. Default: `None` .

Returns

Tensor, vector cross product(s).

Raises

ValueError –when the dimensions of the vector(s) in *a* and/or *b* does not equal 2 or 3.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> output = np.cross(x, y)
>>> print(output)
[[-3  6 -3]
 [ 3 -6  3]]
>>> output = np.cross(x, y, axisc=0)
>>> print(output)
[[-3  3]
 [ 6 -6]
 [-3  3]]
```

13.4.32 mindspore.numpy.cumprod

mindspore.numpy.**cumprod**(*a*, *axis=None*, *dtype=None*)

Returns the cumulative product of elements along a given axis.

Note: Numpy argument *out* is not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) –Input tensor.
- **axis** (*int, optional*) –Axis along which the cumulative product is computed. By default the input is flattened. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor.

Raises

- **TypeError** –If the input can not be converted to tensor or *axis* is not integer.
- **ValueError** –If axis is out of range.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1, 2, 3])
>>> print(np.cumprod(x))
[1 2 6]
```

13.4.33 mindspore.numpy.cumsum

mindspore.numpy.**cumsum**(*a*, *axis=None*, *dtype=None*)

Returns the cumulative sum of the elements along a given axis.

Note: If `a.dtype` is *int8*, *int16* or *bool*, the result *dtype* will be elevated to *int32*.

Parameters

- **a** (`Tensor`) –Input tensor.
- **axis** (*int, optional*) –Axis along which the cumulative sum is computed. The default (`None`) is to compute the cumsum over the flattened array.
- **dtype** (*mindspore.dtype*, optional) –If not specified, stay the same as *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used. Default: `None` .

Returns

Tensor.

Raises

- **TypeError** –If input arguments have types not specified above.
- **ValueError** –If axis is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.cumsum(np.ones((3,3)), axis=0)
>>> print(output)
[[1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]]
```

13.4.34 mindspore.numpy.deg2rad

mindspore.numpy.**deg2rad**(*x*, *dtype=None*)

Converts angles from degrees to radians.

Parameters

- **x** (Tensor) −Angles in degrees.
- **dtype** (*mindspore.dtype*, optional) −Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor, the corresponding angle in radians. This is a tensor scalar if *x* is a tensor scalar.

Raises

TypeError −If *x* is not a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, -5])
>>> output = np.deg2rad(x)
>>> print(output)
[ 0.01745329  0.03490658  0.05235988 -0.06981317 -0.08726647]
```

13.4.35 mindspore.numpy.diff

mindspore.numpy.**diff**(*a*, *n=1*, *axis=- 1*, *prepend=None*, *append=None*)

Calculates the n-th discrete difference along the given axis.

The first difference is given by $out[i] = a[i+1] - a[i]$ along the given axis, higher differences are calculated by using *diff* iteratively.

Note: Since zero-shaped Tensor is not supported in MindSpore, a value error is raised if an empty Tensor is encountered.

Parameters

- **a** (Tensor) −Input tensor.
- **n** (*int, optional*) −The number of times values are differenced. If zero, the input is returned as-is. Default: 1 .
- **axis** (*int, optional*) −The axis along which the difference is taken, default is the last axis. Default: -1 .
- **prepend/append** (Tensor, *optional*) −Values to prepend or append to a along *axis* prior to performing the difference. Scalar values are expanded to arrays with length 1 in the direction of *axis* and the shape of the input array in along all other axes. Otherwise the dimension and shape must match *a* except along axis. Default: None .

Returns

The n-th differences. The shape of the output is the same as a except along *axis* where the dimension is smaller by *n*. The type of the output is the same as the type of the difference between any two elements of *a*. This is the same as the type of *a* in most cases.

Raises

- **TypeError** –If inputs have types not specified above.
- **ValueError** –If `n < 0`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> arr = np.array([1, 3, -1, 0, 4])
>>> print(np.diff(arr, n=2))
[-6  5  3]
```

13.4.36 mindspore.numpy.digitize

`mindspore.numpy.`**digitize**(*x*, *bins*, *right=False*)

Returns the indices of the bins to which each value in input array belongs. If values in *x* are beyond the bounds of *bins*, 0 or `len(bins)` is returned as appropriate.

Parameters

- **x** (*Union[int, float, bool, list, tuple, Tensor]*) –Input array to be binned.
- **bins** (*Union[list, tuple, Tensor]*) –Array of bins. It has to be 1-dimensional and monotonic.
- **right** (*boolean, optional*) –Indicating whether the intervals include the right or the left bin edge. Default behavior is `(right==False)` indicating that the interval does not include the right edge. The left bin end is open in this case, i.e., `bins[i-1] <= x < bins[i]` is the default behavior for monotonically increasing bins.

Returns

Tensor of ints, output array of indices, of same shape as *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = np.array([0, 5, 10, 15, 20])
>>> inds = np.digitize(x, bins)
>>> print(inds)
[1 3 3 4 5]
```

13.4.37 mindspore.numpy.divide

mindspore.numpy.**divide**(*x1*, *x2*, *dtype=None*)

Returns a true division of the inputs, element-wise.

Instead of the Python traditional "floor division", this returns a true division.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –the divident.
- **x2** (Tensor) –the divisor.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2])
>>> x2 = np.full((3, 2), [3, 4])
>>> output = np.divide(x1, x2)
>>> print(output)
[[0.33333334 0.5       ]
[0.33333334 0.5       ]
[0.33333334 0.5       ]]
```

13.4.38 mindspore.numpy.divmod

mindspore.numpy.**divmod**(*x1*, *x2*, *dtype=None*)

Returns element-wise quotient and remainder simultaneously.

Parameters

- **x1** (*Union[Tensor]*) –Dividend tensor.
- **x2** (*Union[Tensor, int, float, bool]*) –Divisor. If x1.shape != x2.shape, they must be broadcastable to a common shape.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Element-wise quotient and remainder from floor division, in format of (quotient, remainder)

Raises

TypeError –If *x1* and *x2* are not Tensor or scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([1, 2, 3, 4, 5])
>>> print(np.divmod(a, 1.5))
(Tensor(shape=[5], dtype=Float32,
 value= [ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00,  2.00000000e+00,  3.
↪00000000e+00]),
 Tensor(shape=[5], dtype=Float32,
 value= [ 1.00000000e+00,  5.00000000e-01,  0.00000000e+00,  1.00000000e+00,  5.00000000e-
↪01]))
```

13.4.39 mindspore.numpy.dot

`mindspore.numpy.`**`dot`**(*a*, *b*)

Returns the dot product of two arrays.

Specifically, If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation). If both *a* and *b* are 2-D arrays, it is matrix multiplication. If either *a* or *b* is 0-D (scalar), it is equivalent to multiply. If *a* is an *N-D* array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*. If *a* is an *N-D* array and *b* is an *M-D* array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*: `dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])`

Note: Numpy argument *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **a** (`Tensor`) – input tensor
- **b** (`Tensor`) – input tensor

Returns

Tensor or scalar, the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned

Raises

`ValueError` – If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.full((1, 3), 7).astype('float32')
>>> b = np.full((2, 3, 4), 5).astype('float32')
>>> output = np.dot(a, b)
>>> print(output)
[[[105. 105. 105. 105.]
[105. 105. 105. 105.]]]
```

13.4.40 mindspore.numpy.ediff1d

mindspore.numpy.**ediff1d**(*ary*, *to_end=None*, *to_begin=None*)

The differences between consecutive elements of a tensor.

Parameters

- **ary** (Tensor) –If necessary, will be flattened before the differences are taken.
- **to_end** (Tensor, *scalar, optional*) –Number(s) to append at the end of the returned differences. Default: None .
- **to_begin** (Tensor, *scalar, optional*) –Number(s) to prepend at the beginning of the returned differences. Default: None .

Returns

The differences.

Raises

TypeError –If inputs have types not specified above.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> arr = np.array([1, 3, -1, 0, 4])
>>> print(np.ediff1d(arr))
[ 2 -4  1  4]
```

13.4.41 mindspore.numpy.exp

mindspore.numpy.**exp**(*x*, *dtype=None*)

Calculates the exponential of all elements in the input array.

Note: Numpy arguments *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. When *where* is provided, *out* must have a tensor value. *out* is not supported for storing the result, however it can be used in combination with *where* to set the value at indices for which *where* is set to False. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, np.float32, np.float64.

Parameters

- **x** (Tensor) –input data.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise exponential of *x*. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.exp(np.arange(5).astype(np.float32))
>>> print(output)
[ 1.         2.718282   7.3890557 20.085537  54.598145 ]
```

13.4.42 mindspore.numpy.exp2

mindspore.numpy.**exp2**(*x*, *dtype=None*)

Calculates 2**p for all p in the input array.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **x** (Tensor) –input values.
- **dtype** (*mindspore.dtype*, optional) –Defaults to None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise 2 to the power *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([2, 3]).astype(np.float32)
>>> output = np.exp2(x)
>>> print(output)
[4. 8.]
```

13.4.43 mindspore.numpy.expm1

mindspore.numpy.**expm1**(*x*, *dtype=None*)

Calculates `exp(x) - 1` for all elements in the array.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **x** (Tensor) – input data.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise exponential minus one, `out = exp(x) - 1`. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.expm1(np.arange(5).astype(np.float32))
>>> print(output)
[ 0.         1.7182819  6.389056  19.085537  53.59815  ]
```

13.4.44 mindspore.numpy.fix

mindspore.numpy.**fix**(*x*)

Rounds to nearest integer towards zero.

Rounds an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

Note: Numpy argument *out* is not supported.

Parameters

x (Tensor) – An array of floats to be rounded.

Returns

Tensor.

Raises

TypeError – If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.fix(np.array([2.1, 2.9, -2.1, -2.9]))
>>> print(output)
[ 2.  2. -2. -2.]
```

13.4.45 mindspore.numpy.float_power

mindspore.numpy.**float_power**(*x1*, *x2*, *dtype=None*)

First array elements raised to powers from second array, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. This differs from the power function in that integers, float16, and float64 are promoted to floats with a minimum precision of float32 so that the result is always inexact. The intent is that the function will return a usable result for negative powers and seldom overflow for positive powers.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Integers and floats are promoted to float32 instead of float64.

Parameters

- **x1** (Tensor) –the bases.
- **x2** (Tensor) –the exponents.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the bases in *x1* raised to the exponents in *x2*. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.arange(6)
>>> x2 = np.array(3)
>>> output = np.float_power(x1, x2)
>>> print(output)
[  0.   1.   8.  27.  64. 125.]
```

13.4.46 mindspore.numpy.floor

mindspore.numpy.**floor**(*x*, *dtype=None*)

Returns the floor of the input, element-wise.

The floor of the scalar *x* is the largest integer *i*, such that `i <= x`.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16 and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **x** (Tensor) –input data.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the floor of each element in *x*. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.floor(np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0]))
>>> print(output)
[-2. -2. -1.  0.  1.  1.  2.]
```

13.4.47 mindspore.numpy.floor_divide

mindspore.numpy.**floor_divide**(*x1*, *x2*, *dtype=None*)

Returns the largest integer smaller or equal to the division of the inputs. It is equivalent to the Python // operator and pairs with the Python % (remainder), function so that `a = a % b + b * (a // b)` up to roundoff.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –Input array.
- **x2** (Tensor) –Input array.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.floor_divide(np.array([1., 2., 3., 4.]), np.array(2.5))
>>> print(output)
[0. 0. 1. 1.]
```

13.4.48 mindspore.numpy.fmod

mindspore.numpy.**fmod**(*x1*, *x2*, *dtype=None*)

Returns the element-wise remainder of division.

This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the dividend *x1*. It is equivalent to the Matlab(TM) rem function and should not be confused with the Python modulus operator `x1 % x2`.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – the first input arrays.
- **x2** (Tensor) – the second input arrays.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the remainder of the division of *x1* by *x2*. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.fmod(np.array([-3, -2, -1, 1, 2, 3]), np.array(2))
>>> print(output)
[-1  0 -1  1  0  1]
```

13.4.49 mindspore.numpy.gcd

mindspore.numpy.**gcd**(*x1*, *x2*, *dtype=None*)

Returns the greatest common divisor of `|x1|` and `|x2|`.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input data.
- **x2** (Tensor) – input data.

- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the greatest common divisor of the absolute value of the inputs. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.gcd(np.arange(6), np.array(20))
>>> print(output)
[20  1  2  1  4  5]
```

13.4.50 mindspore.numpy.gradient

`mindspore.numpy.`**`gradient`**(*f*, **varargs*, *axis=None*, *edge_order=1*)

Returns the gradient of a N-dimensional array. The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries. The returned gradient hence has the same shape as the input array.

Note: Currently we only support *edge_order* =1 and uniform spacing of *varargs*.

Parameters

- **f** (*Union[tuple, list, Tensor]*) –An N-dimensional array containing samples of a scalar function.
- **varargs** (*Union[tuple[number], tuple[tensor scalar]], optional*) –Spacing between f values. Default unitary spacing for all dimensions. Spacing can be specified using: 1. single scalar to specify a sample distance for all dimensions. 2. N scalars to specify a constant sample distance for each dimension.
- **axis** (*Union[None, int, tuple(int), list(int)], optional*) –Gradient is calculated only along the given axis or axes. The default `(axis = None)` is to calculate the gradient for all the axes of the input tensor. *axis* may be negative, in which case it counts from the last to the first *axis*.
- **edge_order** (*int, optional*) –Gradient is calculated using N-th order accurate differences at the boundaries. Default: `1` .

Returns

gradient, a list of tensors (or a single tensor if there is only one dimension to be calculated). Each derivative has the same shape as f.

Raises

- **TypeError** –If the inputs have types not specified above.
- **ValueError** –If *axis* values out of bounds, or shape of *f* has entries < 1.
- **NotImplementedError** –If *edge_order* != 1, or *varargs* contains non-scalar entries.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.gradient([[1, 2, 6], [3, 4, 5]], axis=-1)
>>> print(output)
[[1.  2.5 4. ]
[1.  1.  1. ]]
```

13.4.51 mindspore.numpy.heaviside

mindspore.numpy.**heaviside**(*x1*, *x2*, *dtype=None*)

Computes the Heaviside step function.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) −Input values.
- **x2** (Tensor) −The value of the function when *x1* is 0. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) −Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the output array, element-wise Heaviside step function of *x1*. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.heaviside(np.array([-1.5, 0, 2.0]), np.array(0.5))
>>> print(output)
[0.  0.5 1. ]
>>> output = np.heaviside(np.array([-1.5, 0, 2.0]), np.array(1))
>>> print(output)
[0. 1. 1.]
```

13.4.52 mindspore.numpy.histogram

mindspore.numpy.**histogram**(*a*, *bins=10*, *range=None*, *weights=None*, *density=False*)

Computes the histogram of a dataset.

Note: String values for *bins* is not supported. Deprecated numpy argument *normed* is not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input data. The histogram is computed over the flattened array.
- **bins** (*Union[int, tuple, list, Tensor], optional*) – If *bins* is an int, it defines the number of equal-width bins in the given range (10, by default). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths. Default: `10` .
- **range** (*(float, float), optional*) – The lower and upper range of the bins. If not provided, *range* is simply `(a.min(), a.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. Default: `None` .
- **weights** (*Union[int, float, bool, list, tuple, Tensor], optional*) – An array of weights, of the same shape as *a*. If density is True, the weights are normalized, so that the integral of the density over the range remains 1. Default: `None` .
- **density** (*boolean, optional*) – If False, the result will contain the number of samples in each bin. If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability mass function. Default: `False` .

Returns

(Tensor, Tensor), the values of the histogram and the bin edges.

Raises

ValueError – If *x* and *weights* do not have the same size.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import numpy as np
>>> print(np.histogram([1, 2, 1], bins=[0, 1, 2, 3]))
(Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  2.00000000e+00,  1.
 ↪00000000e+00]),
Tensor(shape=[4], dtype=Int32, value= [0, 1, 2, 3]))
>>> print(np.histogram(np.arange(4), bins=np.arange(5), density=True))
(Tensor(shape=[4], dtype=Float32, value=
[ 2.50000000e-01,  2.50000000e-01,  2.50000000e-01,  2.50000000e-01]),
Tensor(shape=[5], dtype=Int32, value= [0, 1, 2, 3, 4]))
>>> print(np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3]))
(Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  4.00000000e+00,  1.
 ↪00000000e+00]),
Tensor(shape=[4], dtype=Int32, value= [0, 1, 2, 3]))
```

13.4.53 mindspore.numpy.histogram2d

mindspore.numpy.**histogram2d**(*x*, *y*, *bins=10*, *range=None*, *weights=None*, *density=False*)

Computes the multidimensional histogram of some data.

Note: Deprecated numpy argument *normed* is not supported.

Parameters

- **x** (*Union[list, tuple, Tensor]*) – An array with shape *(N,)* containing the x coordinates of the points to be histogrammed.
- **y** (*Union[list, tuple, Tensor]*) – An array with shape *(N,)* containing the y coordinates of the points to be histogrammed.
- **bins** (*Union[int, tuple, list], optional*) – Default: `10` . The bin specification:

  If int, the number of bins for the two dimensions `(nx=ny=bins)`.

  If array_like, the bin edges for the two dimensions `(x_edges=y_edges=bins)`.

  If [int, int], the number of bins in each dimension `(nx, ny = bins)`.

  If [array, array], the bin edges in each dimension `(x_edges, y_edges = bins)`.

  A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.
- **range** (*Union[list, tuple], optional*) – has shape (2, 2), the leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the bins parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram. Default: `None` .
- **weights** (*Union[list, tuple, Tensor], optional*) – An array with shape *(N,)* of values *w_i* weighing each sample *(x_i, y_i)*. Default: `None` .
- **density** (*boolean, optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability density function at the bin, `bin_count / sample_count / bin_volume`. Default: `False` .

Returns

(Tensor, Tensor, Tensor), the values of the bi-directional histogram and the bin edges along the first and second dimensions.

Raises

ValueError – If *range* does not have the same size as the number of samples.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import numpy as np
>>> x = np.arange(5)
>>> y = np.arange(2, 7)
>>> print(np.histogram2d(x, y, bins=(2, 3)))
(Tensor(shape=[2, 3], dtype=Float32, value=
[[ 2.00000000e+00,  0.00000000e+00,  0.00000000e+00],
[ 0.00000000e+00,  1.00000000e+00,  2.00000000e+00]]),
Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  2.00000000e+00,  4.00000000e+00]),
```

(continues on next page)

(continued from previous page)

```
Tensor(shape=[4], dtype=Float32, value=
[ 2.00000000e+00,  3.33333349e+00,  4.66666698e+00,  6.00000000e+00]))
```

13.4.54 mindspore.numpy.histogramdd

mindspore.numpy.**histogramdd**(*sample*, *bins=10*, *range=None*, *weights=None*, *density=False*)

Computes the multidimensional histogram of some data.

Note: Deprecated numpy argument *normed* is not supported.

Parameters

- **sample** (*Union[list, tuple, Tensor]*) – The data to be histogrammed, either *(N, D)* array, or *(D, N)* array_like. Note the unusual interpretation of sample when an array_like:

  When an array, each row is a coordinate in a *D-dimensional* space, such as `histogramdd(np.array([p1, p2, p3]))`.

  When an array_like, each element is the list of values for single coordinate, such as `histogramdd((X, Y, Z))`.

  The first form should be preferred.

- **bins** (*Union[int, tuple, list], optional*) – Default: `10` . The bin specification:

  A sequence of arrays describing the monotonically increasing bin edges along each dimension.

  The number of bins for each dimension `(nx, ny, ···=bins)`

  The number of bins for all dimensions `(nx=ny=···=bins)`.

- **range** (*Union[list, tuple], optional*) – A sequence of length *D*, each an optional `(lower, upper)` tuple giving the outer bin edges to be used if the edges are not given explicitly in bins. An entry of None in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, None, is equivalent to passing a tuple of *D* None values. Default: `None` .

- **weights** (*Union[list, tuple, Tensor], optional*) – An array with shape *(N,)* of values *w_i* weighing each sample `(x_i, y_i, z_i, ···)`. Default: `None` .

- **density** (*boolean, optional*) – If False, the default, returns the number of samples in each bin. If True, returns the probability density function at the bin, `bin_count / sample_count / bin_volume`. Default: `False` .

Returns

(Tensor, list of Tensor), the values of the histogram and the bin edges.

Raises

ValueError – If *range* does not have the same size as the number of samples.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import numpy as np
>>> sample = np.arange(15).reshape(5, 3)
>>> print(sample)
[[ 0  1  2]
[ 3  4  5]
[ 6  7  8]
[ 9 10 11]
[12 13 14]]
>>> print(np.histogramdd(sample, bins=(2, 3, 4)))
(Tensor(shape=[2, 3, 4], dtype=Float32, value=
[[[ 1.00000000e+00,  1.00000000e+00,  0.00000000e+00,  0.00000000e+00],
[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+00]],
[[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
[ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,  2.00000000e+00]]]),
[Tensor(shape=[3], dtype=Float32, value= [ 0.00000000e+00,  6.00000000e+00,  1.
↪20000000e+01]),
Tensor(shape=[4], dtype=Float32, value=
[ 1.00000000e+00,  5.00000000e+00,  9.00000000e+00,  1.30000000e+01]),
Tensor(shape=[5], dtype=Float32, value=
[ 2.00000000e+00,  5.00000000e+00,  8.00000000e+00,  1.10000000e+01,  1.40000000e+01])])
```

13.4.55 mindspore.numpy.hypot

`mindspore.numpy.`**`hypot`**(*x1*, *x2*, *dtype=None*)

Given the "legs" of a right triangle, returns its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If *x1* or *x2* is scalar_like (i.e., unambiguously cast-able to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16 and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **x1** (`Tensor`) –Leg of the triangle(s).
- **x2** (`Tensor`) –Leg of the triangle(s). If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the hypotenuse of the triangle(s). This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
>>> print(output)
[[5. 5. 5.]
[5. 5. 5.]
[5. 5. 5.]]
>>> output = np.hypot(3*np.ones((3, 3)), np.array([4.0]))
>>> print(output)
[[5. 5. 5.]
[5. 5. 5.]
[5. 5. 5.]]
```

13.4.56 mindspore.numpy.inner

mindspore.numpy.**inner**(*a*, *b*)

Returns the inner product of two tensors.

Ordinary inner product of vectors for 1-D tensors (without complex conjugation), in higher dimensions a sum product over the last axes.

Note: Numpy argument *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **a** (Tensor) – input tensor. If *a* and *b* are nonscalar, their last dimensions must match.
- **b** (Tensor) – input tensor. If *a* and *b* are nonscalar, their last dimensions must match.

Returns

Tensor or scalar.

Raises

ValueError – If `x1.shape[-1] != x2.shape[-1]`.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((5, 3))
>>> b = np.ones((2, 7, 3))
>>> output = np.inner(a, b)
>>> print(output)
[[[3. 3. 3. 3. 3. 3. 3.]
[3. 3. 3. 3. 3. 3. 3.]]
[[3. 3. 3. 3. 3. 3. 3.]
[3. 3. 3. 3. 3. 3. 3.]]
```

(continues on next page)

(continued from previous page)

```
[[3. 3. 3. 3. 3. 3. 3.]
[3. 3. 3. 3. 3. 3. 3.]]
[[3. 3. 3. 3. 3. 3. 3.]
[3. 3. 3. 3. 3. 3. 3.]]
[[3. 3. 3. 3. 3. 3. 3.]
[3. 3. 3. 3. 3. 3. 3.]]]
```

13.4.57 mindspore.numpy.interp

mindspore.numpy.**interp**(*x*, *xp*, *fp*, *left=None*, *right=None*)

One-dimensional linear interpolation for monotonically increasing sample points. Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points *(xp, fp)*, evaluated at *x*.

Note: Numpy argument *period* is not supported. Complex values are not supported.

Parameters

- **x** (*Union[int, float, bool, list, tuple, Tensor]*) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (*Union[int, float, bool, list, tuple, Tensor]*) – 1-D sequence of floats, the x-coordinates of the data points, must be increasing.
- **fp** (*Union[int, float, bool, list, tuple, Tensor]*) – 1-D sequence of floats, the y-coordinates of the data points, same length as *xp*.
- **left** (*float, optional*) – Value to return for `x < xp[0]`, default is `fp[0]` once obtained. Default: `None` .
- **right** (*float, optional*) – Value to return for `x > xp[-1]`, default is `fp[-1]` once obtained. Default: `None` .

Returns

Tensor, the interpolated values, same shape as *x*.

Raises

ValueError – If *xp* or *fp* is not one-dimensional, or if *xp* and *fp* do not have the same length.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> print(np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp))
[3.         3.         2.5        0.55999994 0.        ]
>>> UNDEF = -99.0
>>> print(np.interp(3.14, xp, fp, right=UNDEF))
-99.0
```

13.4.58 mindspore.numpy.invert

mindspore.numpy.**invert**(*x*, *dtype=None*)

Computes bit-wise inversion, or bit-wise NOT, element-wise. Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator ~. For signed integer inputs, the two's complement is returned. In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two's-complement system can represent every integer in the range -2^{N-1} to +2^{N-1}-1.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Supported dtypes on Ascend: np.int16, np.uint16.

Parameters

- **x** (Tensor) –Only integer and boolean types are handled.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore.numpy as np
>>> print(np.invert(np.array(13, dtype=np.uint16)))
65522
```

13.4.59 mindspore.numpy.kron

mindspore.numpy.**kron**(*a*, *b*)

Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

Note: Booleans are not supported.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) –input values.
- **b** (*Union[int, float, list, tuple, Tensor]*) –input values.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.kron([1,10,100], [5,6,7])
>>> print(output)
[  5   6   7  50  60  70 500 600 700]
>>> output = np.kron([5,6,7], [1,10,100])
>>> print(output)
[  5  50 500   6  60 600   7  70 700]
>>> output = np.kron(np.eye(2), np.ones((2,2)))
>>> print(output)
[[1. 1. 0. 0.]
[1. 1. 0. 0.]
[0. 0. 1. 1.]
[0. 0. 1. 1.]]
```

13.4.60 mindspore.numpy.lcm

mindspore.numpy.**lcm**(*x1*, *x2*, *dtype=None*)

Returns the lowest common multiple of |x1| and |x2|.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input data.
- **x2** (Tensor) – input data.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the lowest common multiple of the absolute value of the inputs. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.lcm(np.arange(6), np.array(20))
>>> print(output)
[ 0 20 20 60 20 20]
```

13.4.61 mindspore.numpy.log

mindspore.numpy.**log**(*x*, *dtype=None*)

Returns the natural logarithm, element-wise.

The natural logarithm log is the inverse of the exponential function, so that `log(exp(x)) = x`. The natural logarithm is logarithm in base e.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **x** (Tensor) –Input array.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the natural logarithm of *x*, element-wise. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([2, 3, 4]).astype('float32')
>>> output = np.log(x)
>>> print(output)
[0.69314575 1.09861    1.3862929 ]
```

13.4.62 mindspore.numpy.log10

mindspore.numpy.**log10**(*x*, *dtype=None*)

Base-10 logarithm of *x*.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([10, 100, 1000]).astype('float16')
>>> output = np.log10(x)
>>> print(output)
[1. 2. 3.]
```

13.4.63 mindspore.numpy.log1p

mindspore.numpy.**log1p**(*x*, *dtype=None*)

Returns the natural logarithm of one plus the input array, element-wise.

Calculates `log(1 + x)`.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input array.
- **dtype** (*mindspore.dtype*) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([1, 2, 3]).astype('float16')
>>> output = np.log1p(x)
>>> print(output)
[0.6934 1.099 1.387 ]
```

13.4.64 mindspore.numpy.log2

mindspore.numpy.**log2**(*x*, *dtype=None*)

Base-2 logarithm of *x*.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([2, 4, 8]).astype('float16')
>>> output = np.log2(x)
>>> print(output)
[1. 2. 3.]
```

13.4.65 mindspore.numpy.logaddexp

`mindspore.numpy.`**`logaddexp`**(*x1*, *x2*, *dtype=None*)

Logarithm of the sum of exponentiations of the inputs. Calculates `log(exp(x1) + exp(x2))`.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –Input array.
- **x2** (`Tensor`) –Input array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([1, 2, 3]).astype('float16')
>>> x2 = np.array(2).astype('float16')
>>> output = np.logaddexp(x1, x2)
>>> print(output)
[2.312 2.693 3.312]
```

13.4.66 mindspore.numpy.logaddexp2

mindspore.numpy.**logaddexp2**(*x1*, *x2*, *dtype=None*)

Logarithm of the sum of exponentiations of the inputs in base of 2.

Calculates `log2(2**x1 + 2**x2)`. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) –Input tensor.
- **x2** (Tensor) –Input tensor. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.array([2, 4, 8]).astype('float16')
>>> x2 = np.array(2).astype('float16')
>>> output = np.logaddexp2(x1, x2)
>>> print(output)
[3. 4.32 8.02]
```

13.4.67 mindspore.numpy.matmul

mindspore.numpy.**matmul**(*x1*, *x2*, *dtype=None*)

Returns the matrix product of two arrays.

Note: Numpy arguments *out*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16 and np.float32. On CPU, the supported dtypes are np.float16 and np.float32.

Parameters

- **x1** (Tensor) –Input tensor, scalar not allowed.
- **x2** (Tensor) –Input tensor, scalar not allowed.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the matrix product of the inputs. This is a scalar only when both *x1*, *x2* are 1-d vectors.

Raises

ValueError – If the last dimension of *x1* is not the same size as the second-to-last dimension of *x2*, or if a scalar value is passed in.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.arange(2*3*4).reshape(2, 3, 4).astype('float32')
>>> x2 = np.arange(4*5).reshape(4, 5).astype('float32')
>>> output = np.matmul(x1, x2)
>>> print(output)
[[[  70.   76.   82.   88.   94.]
[ 190.  212.  234.  256.  278.]
[ 310.  348.  386.  424.  462.]]
[[ 430.  484.  538.  592.  646.]
[ 550.  620.  690.  760.  830.]
[ 670.  756.  842.  928. 1014.]]]
```

13.4.68 mindspore.numpy.matrix_power

mindspore.numpy.**matrix_power**(*a*, *n*)

Raises a square matrix to the (integer) power *n*.

For positive integers *n*, the power is computed by repeated matrix squarings and matrix multiplications. If $n == 0$, the identity matrix of the same shape as *M* is returned.

Note: Stacks of object matrices are not currently supported and $n < 0$ is not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) – Input matrix.
- **n** (*int*) – The exponent can be any integer or long integer, positive or zero.

Returns

Tensor.

Raises

- **TypeError** – If the input can not be converted to a tensor or the exponent is not integer.
- **ValueError** – If the input includes less than 2 dimensions or the last 2 dimensions are not square.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import numpy as np
>>> a = np.arange(16).reshape(4, 4).astype('float32')
>>> print(np.matrix_power(a, 2))
[[ 56.  62.  68.  74.]
 [152. 174. 196. 218.]
 [248. 286. 324. 362.]
 [344. 398. 452. 506.]]
```

13.4.69 mindspore.numpy.maximum

mindspore.numpy.**maximum**(*x1*, *x2*, *dtype=None*)

Returns the element-wise maximum of array elements.

Compares two arrays and returns a new array containing the element-wise maxima.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On Ascend, input arrays containing inf or NaN are not supported.

Parameters

- **x1** (Tensor) – Input array
- **x2** (Tensor) – The array holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the maximum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.maximum(np.array([2, 3, 4]), np.array([1, 5, 2]))
>>> print(output)
[2 5 4]
```

13.4.70 mindspore.numpy.mean

mindspore.numpy.**mean**(*a*, *axis=None*, *keepdims=False*, *dtype=None*)

Computes the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **a** (Tensor) – input tensor containing numbers whose mean is desired. If a is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple(int), None], optional*) – Axis or axes along which the means are computed. The default is to compute the mean of the flattened array. If this is a tuple of ints, a mean is performed over multiple axes. Default: None .
- **keepdims** (*bool, optional*) – If this is set to True , the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor. Default: False .
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, an array containing the mean values.

Raises

ValueError – If axes are out of the range of [-a.ndim, a.ndim), or if the axes contain duplicates.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(6, dtype='float32')
>>> output = np.mean(a, 0)
>>> print(output)
2.5
```

13.4.71 mindspore.numpy.minimum

mindspore.numpy.**minimum**(*x1*, *x2*, *dtype=None*)

Element-wise minimum of tensor elements.

Compares two tensors and returns a new tensor containing the element-wise minima.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On Ascend, input arrays containing inf or NaN are not supported.

Parameters

- **x1** (Tensor) –first input tensor to be compared.
- **x2** (Tensor) –second input tensor to be compared.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor, element-wise minimum of *x1* and *x2*.

Raises

- **TypeError** –If inputs have types not specified above.
- **ValueError** –If the shapes of *x1* and *x2* cannot be broadcast.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.asarray([1, 2])
>>> b = np.asarray([[1, 3],[1, 4]])
>>> print(np.minimum(a, b))
[[1 2]
[1 2]]
```

13.4.72 mindspore.numpy.multi_dot

mindspore.numpy.**multi_dot**(*arrays*)

Computes the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order. multi_dot chains numpy.dot and uses optimal parenthesization of the matrices. For more information, refer to the wiki page. Depending on the shapes of the matrices, this can speed up the multiplication a lot. If the first argument is 1-D, it is treated as a row vector. If the last argument is 1-D, it is treated as a column vector. The other arguments must be 2-D.

Note: Numpy argument *out* is not supported.

Parameters

arrays (*sequence of array_like*) –If the first argument is 1-D, it is treated as row vector. If the last argument is 1-D, it is treated as column vector. The other arguments must be 2-D.

Returns

Tensor, the dot product of the supplied arrays.

Raises

ValueError –arrays are not 2-D.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> A = np.ones((10000, 100))
>>> B = np.ones((100, 100))
>>> C = np.ones((100, 5))
>>> D = np.ones((5, 333))
>>> output = np.multi_dot([A, B, C, D])
>>> print(output)
[[50000. 50000. 50000. ... 50000. 50000. 50000.]
[50000. 50000. 50000. ... 50000. 50000. 50000.]
[50000. 50000. 50000. ... 50000. 50000. 50000.]
...
[50000. 50000. 50000. ... 50000. 50000. 50000.]
[50000. 50000. 50000. ... 50000. 50000. 50000.]
[50000. 50000. 50000. ... 50000. 50000. 50000.]]
```

13.4.73 mindspore.numpy.multiply

mindspore.numpy.**multiply**(*x1*, *x2*, *dtype=None*)

Multiplies arguments element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input tensor to be multiplied.
- **x2** (Tensor) – input tensor to be multiplied.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the product of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2])
>>> x2 = np.full((3, 2), [3, 4])
>>> output = np.multiply(x1, x2)
>>> print(output)
[[3 8]
[3 8]
[3 8]]
```

13.4.74 mindspore.numpy.nancumsum

mindspore.numpy.**nancumsum**(*a*, *axis=None*, *dtype=None*)

Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. The cumulative sum does not change when NaNs are encountered and leading NaNs are replaced by zeros.

Zeros are returned for slices that are all-NaN or empty.

Note: If `a.dtype` is *int8*, *int16* or *bool*, the result *dtype* will be elevated to *int32*.

Parameters

- **a** (`Tensor`) – Input tensor.
- **axis** (*int, optional*) – Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.
- **dtype** (*mindspore.dtype*, optional) – If not specified, stay the same as *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used. Default: `None`.

Returns

Tensor.

Raises

- **TypeError** – If input arguments have types not specified above.
- **ValueError** – If axis is out of range.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, 2], [3, np.nan]])
>>> output = np.nancumsum(a)
>>> print(output)
[1. 3. 6. 6.]
>>> output = np.nancumsum(a, axis=0)
>>> print(output)
[[1. 2.]
[4. 2.]]
>>> output = np.nancumsum(a, axis=1)
>>> print(output)
[[1. 3.]
[3. 3.]]
```

13.4.75 mindspore.numpy.nanmax

mindspore.numpy.**nanmax**(*a*, *axis=None*, *dtype=None*, *keepdims=False*)

Return the maximum of an array or maximum along an axis, ignoring any NaNs.

Note: Numpy arguments *out* is not supported. For all NaN slices, a very small negative number is returned instead of NaN.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) – Array containing numbers whose maximum is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple(int), None], optional*) – Axis or axes along which the maximum is computed. The default is to compute the maximum of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.
- **keepdims** (*boolean, optional*) – Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError – If axes are out of the range of `[-a.ndim, a.ndim)` , or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, 2], [3, np.nan]])
>>> output = np.nanmax(a)
>>> print(output)
3.0
>>> output = np.nanmax(a, axis=0)
>>> print(output)
[3. 2.]
```

13.4.76 mindspore.numpy.nanmean

mindspore.numpy.**nanmean**(*a*, *axis=None*, *dtype=None*, *keepdims=False*)

Computes the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float32 intermediate and return values are used for integer inputs.

Note: Numpy arguments *out* is not supported.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) –Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple of int, None], optional*) –Axis or axes along which the mean is computed. The default is to compute the mean of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.
- **keepdims** (*boolean, optional*) –Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError –If axes are out of the range of `[-a.ndim, a.ndim)` , or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, np.nan], [3, 4]])
>>> output = np.nanmean(a)
>>> print(output)
2.6666667
>>> output = np.nanmean(a, axis=0)
>>> print(output)
[2. 4.]
>>> output = np.nanmean(a, axis=1)
>>> print(output)
[1.  3.5]
```

13.4.77 mindspore.numpy.nanmin

mindspore.numpy.**nanmin**(*a*, *axis=None*, *dtype=None*, *keepdims=False*)

Returns the minimum of array elements over a given axis, ignoring any NaNs.

Note: Numpy arguments *out* is not supported. For all-NaN slices, a very large number is returned instead of NaN. On Ascend, since checking for NaN is currently not supported, it is not recommended to use np.nanmin. If the array does not contain NaN, np.min should be used instead.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) –Array containing numbers whose minimum is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple(int), None], optional*) –Axis or axes along which the minimum is computed. The default is to compute the minimum of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

- **keepdims** (*boolean, optional*) –Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError –If axes are out of the range of `[-a.ndim, a.ndim)` , or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, 2], [3, np.nan]])
>>> output = np.nanmin(a)
>>> print(output)
1.0
>>> output = np.nanmin(a, axis=0)
>>> print(output)
[1. 2.]
```

13.4.78 mindspore.numpy.nanstd

`mindspore.numpy.`**nanstd**(*a*, *axis=None*, *dtype=None*, *ddof=0*, *keepdims=False*)

Computes the standard deviation along the specified axis, while ignoring NaNs.

Returns the standard deviation, a measure of the spread of a distribution, of the non-NaN array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) –Calculates the standard deviation of the non-NaN values.
- **axis** (*Union[int, tuple of int, None], optional*) –Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.
- **ddof** (*int, optional*) –"Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where *N* represents the number of non-NaN elements. By default *ddof* is zero.
- **keepdims** (*boolean, optional*) –Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError –If axes are out of the range of `[-a.ndim, a.ndim)` , or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, np.nan], [3, 4]])
>>> output = np.nanstd(a)
>>> print(output)
1.2472192
>>> output = np.nanstd(a, axis=0)
>>> print(output)
[1. 0.]
>>> output = np.nanstd(a, axis=1)
>>> print(output)
[0.  0.5]
```

13.4.79 mindspore.numpy.nansum

mindspore.numpy.**nansum**(*a*, *axis=None*, *dtype=None*, *keepdims=False*)

Returns the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

Note: Numpy arguments *out* is not supported.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) – Array containing numbers whose sum is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple of int, None], optional*) – Axis or axes along which the sum is computed. The default is to compute the sum of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.
- **keepdims** (*boolean, optional*) – Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError – If axes are out of the range of `[-a.ndim, a.ndim)` , or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, 1], [1, np.nan]])
>>> output = np.nansum(a)
>>> print(output)
3.0
>>> output = np.nansum(a, axis=0)
>>> print(output)
[2. 1.]
```

13.4.80 mindspore.numpy.nanvar

mindspore.numpy.**nanvar**(*a*, *axis=None*, *dtype=None*, *ddof=0*, *keepdims=False*)

Computes the variance along the specified axis, while ignoring NaNs.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *out* is not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **a** (*Union[int, float, list, tuple, Tensor]*) – Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.
- **axis** (*Union[int, tuple of int, None], optional*) – Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array. Default: `None` .
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.
- **ddof** (*int, optional*) – "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where *N* represents the number of non-NaN elements. By default *ddof* is zero.
- **keepdims** (*boolean, optional*) – Default: `False` . If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

Returns

Tensor.

Raises

ValueError – If axes are out of the range of `[-a.ndim, a.ndim)`, or if the axes contain duplicates.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.array([[1, np.nan], [3, 4]])
>>> output = np.nanvar(a)
>>> print(output)
1.5555557
>>> output = np.nanvar(a, axis=0)
>>> print(output)
[1. 0.]
>>> output = np.nanvar(a, axis=1)
>>> print(output)
[0.   0.25]
```

13.4.81 mindspore.numpy.negative

mindspore.numpy.**negative**(*a*, *dtype=None*)

Numerical negative, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **a** (Tensor) −Input tensor.
- **dtype** (*mindspore.dtype*, optional) −Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.asarray([1, -1]).astype('float32')
>>> output = np.negative(a)
>>> print(output)
[-1. 1.]
```

13.4.82 mindspore.numpy.norm

mindspore.numpy.**norm**(*x*, *ord=None*, *axis=None*, *keepdims=False*)

Matrix or vector norm. This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the ord parameter.

Note: Nuclear norm and 2-norm are not supported for matrices.

Parameters

- **x** (*Union[int, float, bool, list, tuple, Tensor]*) – Input array. If *axis* is None, *x* must be 1-D or 2-D, unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.
- **ord** (*Union[None, 'fro', 'nuc', inf, -inf, int, float], optional*) – Order of the norm. inf means numpy's inf object. Default: `None` .
- **axis** (*Union[int, 2-tuple(int), None], optional*) – If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when x is 1-D) or a matrix norm (when *x* is 2-D) is returned. The default is None.
- **keepdims** (*boolean, optional*) – If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original *x*.

Returns

Tensor, norm of the matrix or vector(s).

Raises

ValueError – If the norm order is not defined.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.norm(np.arange(9).astype(np.float32)))
14.282857
```

13.4.83 mindspore.numpy.outer

mindspore.numpy.**outer**(*a*, *b*)

Computes the outer product of two vectors.

Given two vectors, `a = [a0, a1, ..., aM]` and `b = [b0, b1, ..., bN]`, the outer product is:

```
[[a0*b0 a0*b1 ... a0*bN ]
[a1*b0 . ]
[ ... . ]
[aM*b0 aM*bN ]]
```

Note: Numpy argument `out` is not supported. On GPU, the supported dtypes are np.float16, and np.float32. On CPU, the supported dtypes are np.float16, np.float32, and np.float64.

Parameters

- **a** (Tensor) – first input vector. Input is flattened if not already 1-dimensional.
- **b** (Tensor) – second input vector. Input is flattened if not already 1-dimensional.

Returns

Tensor or scalar, `out[i, j] = a[i] * b[j]`.

Raises

TypeError – If the input is not a tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.full(7, 2).astype('float32')
>>> b = np.full(4, 3).astype('float32')
>>> output = np.outer(a, b)
>>> print(output)
[[6. 6. 6. 6.]
[6. 6. 6. 6.]
[6. 6. 6. 6.]
[6. 6. 6. 6.]
[6. 6. 6. 6.]
[6. 6. 6. 6.]
[6. 6. 6. 6.]]
```

13.4.84 mindspore.numpy.polyadd

mindspore.numpy.**polyadd**(*a1*, *a2*)

Finds the sum of two polynomials. Returns the polynomial resulting from the sum of two input polynomials.

Note: Numpy object poly1d is currently not supported.

Parameters

- **a1** (*Union[int, float, list, tuple, Tensor]*) – Input polynomial.
- **a2** (*Union[int, float, list, tuple, Tensor]*) – Input polynomial.

Returns

Tensor, the sum of the inputs.

Raises

ValueError –If the input array has more than 1 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polyadd([1, 2], [9, 5, 4]))
[9 6 6]
```

13.4.85 mindspore.numpy.polyder

mindspore.numpy.**polyder**(*p*, *m=1*)

Returns the derivative of the specified order of a polynomial.

Note: Numpy object poly1d is currently not supported.

Parameters

- **p** (*Union[int, float, bool, list, tuple, Tensor]*) –Polynomial to differentiate. A sequence is interpreted as polynomial coefficients.
- **m** (*int, optional*) –Default: `1` , order of differentiation.

Returns

Tensor, a new polynomial representing the derivative.

Raises

ValueError –If *p* has more than 1 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polyder([1, 1, 1, 1]))
[3 2 1]
```

13.4.86 mindspore.numpy.polyint

mindspore.numpy.**polyint**(*p*, *m=1*, *k=None*)

Returns an antiderivative (indefinite integral) of a polynomial.

Note: Numpy object poly1d is currently not supported.

Parameters

- **p** (*Union[int, float, bool, list, tuple, Tensor]*) –Polynomial to integrate. A sequence is interpreted as polynomial coefficients.
- **m** (*int, optional*) –Defaults to 1, Order of the antiderivative.
- **k** (*Union[int, list[int]], optional*) –Integration constants. They are given in the order of integration: those corresponding to highest-order terms come first. If None (default), all constants are assumed to be zero. If `m = 1`, a single scalar can be given instead of a list.

Returns

Tensor, a new polynomial representing the antiderivative.

Raises

ValueError –If *p* has more than 1 dimensions.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polyint([1, 1, 1]))
[0.33333334 0.5        1.         0.        ]
```

13.4.87 mindspore.numpy.polymul

mindspore.numpy.**polymul**(*a1*, *a2*)

Finds the product of two polynomials.

Note: Numpy object poly1d is currently not supported.

Parameters

- **a1** (*Union[int, float, bool, list, tuple, Tensor]*) –Input polynomial.
- **a2** (*Union[int, float, bool, list, tuple, Tensor]*) –Input polynomial.

Returns

Tensor, a new polynomial representing the derivative.

Raises

ValueError –If the input array has more than 1 dimensions.

Supported Platforms:

GPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polymul([3, 1, 2], [2, 5]))
[ 6 17  9 10]
```

13.4.88 mindspore.numpy.polysub

mindspore.numpy.**polysub**(*a1*, *a2*)

Difference (subtraction) of two polynomials. Given two polynomials *a1* and *a2*, returns `a1 - a2`.

Note: Numpy object poly1d is currently not supported.

Parameters

- **a1** (*Union[int, float, list, tuple, Tensor]*) –Minuend polynomial.
- **a2** (*Union[int, float, list, tuple, Tensor]*) –Subtrahend polynomial.

Returns

Tensor, the difference of the inputs.

Raises

ValueError –If the input array has more than 1 dimensions.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polysub([2, 10, -2], [3, 10, -4]))
[-1  0  2]
```

13.4.89 mindspore.numpy.polyval

mindspore.numpy.**polyval**(*p*, *x*)

Evaluates a polynomial at specific values. If *p* is of length *N*, this function returns the value: `p[0]*x**(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x + p[N-1]` If *x* is a sequence, then `p(x)` is returned for each element of *x*. If *x* is another polynomial then the composite polynomial `p(x(t))` is returned.

Note: Numpy object poly1d is currently not supported.

Parameters

- **p** (*Union[int, float, bool, list, tuple, Tensor]*) – 1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term.
- **x** (*Union[int, float, bool, list, tuple, Tensor]*) – A number, an array of numbers, at which to evaluate *p*.

Returns

Tensor.

Raises

ValueError – If *p* has more than 1 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.polyval([3.,0.,1.], 5.))
76.0
```

13.4.90 mindspore.numpy.positive

mindspore.numpy.**positive**(*a*, *dtype=None*)

Numerical positive, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **a** (`Tensor`) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> a = np.asarray([1, -1]).astype('float32')
>>> output = np.positive(a)
>>> print(output)
[1. -1.]
```

13.4.91 mindspore.numpy.power

mindspore.numpy.**power**(*x1*, *x2*, *dtype=None*)

First array elements raised to powers from second array, element-wise.

Raises each base in *x1* to the positionally-corresponding power in *x2*.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16, and np.float32.

Parameters

- **x1** (Tensor) – The bases.
- **x2** (Tensor) – The exponents.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the bases in *x1* raised to the exponents in *x2*. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2]).astype('float32')
>>> x2 = np.full((3, 2), [3, 4]).astype('float32')
>>> output = np.power(x1, x2)
>>> print(output)
[[ 1. 16.]
 [ 1. 16.]
 [ 1. 16.]]
```

13.4.92 mindspore.numpy.promote_types

mindspore.numpy.**promote_types**(*type1*, *type2*)

Returns the data type with the smallest size and smallest scalar kind.

Note: The promotion rule is slightly different from original Numpy, but more like jax, due to the preference on `32-bit` over `64-bit` data types.

Parameters

- **type1** (Union[*mindspore.dtype*, str]) –First data type.
- **type2** (Union[*mindspore.dtype*, str]) –Second data type.

Returns

The promoted data type.

Raises

TypeError –If the input are not valid *mindspore.dtype* input.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.promote_types(np.float32, np.float64)
>>> print(output)
Float64
```

13.4.93 mindspore.numpy.ptp

mindspore.numpy.**ptp**(*x*, *axis=None*, *keepdims=False*)

Range of values (maximum - minimum) along an axis. The name of the function comes from the acronym for "peak to peak".

Note: Numpy arguments *dtype* and *out* are not supported.

Parameters

- **x** (`Tensor`) –Input tensor.
- **axis** (*Union[None, int, tuple(int)]*) –Axis or axes along which the range is computed. The default is to compute the variance of the flattened array. Default: `None`.
- **keepdims** (*bool*) –If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor. If the default value is passed, then keepdims will not be passed through to the ptp method of sub-classes of tensor, however any non-default value will be. Default: `False`.

Returns

Tensor.

Raises

TypeError –If inputs have types not specified above.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([[4.0, 9.0, 2.0, 10.0], [6.0, 9.0, 7.0, 12.0]])
>>> print(np.ptp(x, axis=1))
[8. 6.]
>>> print(np.ptp(x, axis=0))
[2. 0. 5. 2.]
```

13.4.94 mindspore.numpy.rad2deg

mindspore.numpy.**rad2deg**(*x*, *dtype=None*)

Converts angles from radians to degrees.

Parameters

- **x** (Tensor) –Angles in radians.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor, the corresponding angle in degrees. This is a tensor scalar if *x* is a tensor scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, -5])
>>> output = np.rad2deg(x)
>>> print(output)
[  57.295776  114.59155   171.88733  -229.1831   -286.47888 ]
```

13.4.95 mindspore.numpy.radians

mindspore.numpy.**radians**(*x*, *dtype=None*)

Converts angles from degrees to radians.

Parameters

- **x** (Tensor) −Angles in degrees.
- **dtype** (*mindspore.dtype*, optional) −Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor, the corresponding radian values. This is a tensor scalar if *x* is a tensor scalar.

Raises

TypeError −If *x* is not a tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.asarray([1, 2, 3, -4, -5])
>>> output = np.radians(x)
>>> print(output)
[ 0.01745329  0.03490658  0.05235988 -0.06981317 -0.08726647]
```

13.4.96 mindspore.numpy.ravel_multi_index

mindspore.numpy.**ravel_multi_index**(*multi_index*, *dims*, *mode='clip'*, *order='C'*)

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

Note: *raise* mode is not supported. Default mode is *clip*.

Parameters

- **multi_index** (*tuple of array_like*) −A tuple of integer arrays, one array for each dimension.
- **dims** (*Union[int, tuple(int)]*) −The shape of array into which the indices from multi_index apply.
- **mode** ({*wrap*, *clip*}, optional) −Specifies how out-of-bounds indices are handled. Default: 'clip'.
  - *wrap*: wrap around
  - *clip*: clip to the range

  In *clip* mode, a negative index which would normally wrap will clip to 0 instead.
- **order** ({*C*, *F*}, optional) −Determines whether the multi-index should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

Returns

Raveled_indices array. An array of indices into the flattened version of an array of dimensions dims.

Raises

- **TypeError** –If *multi_index* or *dims* can not be converted to tensor or *dims* is not a sequence of integer values.
- **ValueError** –If the length of *multi_index* and that of *dims* are not equal.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore.numpy as np
>>> arr = np.array([[3, 6, 6], [4, 5, 1]])
>>> output = np.ravel_multi_index(arr, (7, 6))
>>> print(output)
[22. 41. 37.]
>>> output = np.ravel_multi_index((3, 1, 4, 1), (6, 7, 8, 9))
>>> print(output)
1621.0
```

13.4.97 mindspore.numpy.reciprocal

`mindspore.numpy.`**`reciprocal`**(*x*, *dtype=None*)

Returns the reciprocal of the argument, element-wise.

Calculates `1/x`.

Note: Numpy arguments *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. When *where* is provided, *out* must have a tensor value. *out* is not supported for storing the result, however it can be used in combination with *where* to set the value at indices for which *where* is set to False.

Parameters

- **x** (`Tensor`) –Input array. For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(1, 7).reshape(2, 3).astype('float32')
>>> output = np.reciprocal(x)
>>> print(output)
[[1.         0.5        0.33333334]
[0.25       0.2        0.16666667]]
```

13.4.98 mindspore.numpy.remainder

mindspore.numpy.**remainder**(*x1*, *x2*, *dtype=None*)

Returns element-wise remainder of division.

Computes the remainder complementary to the floor_divide function. It is equivalent to the Python modulus operator `x1 % x2` and has the same sign as the divisor *x2*. The MATLAB function equivalent to np.remainder is mod.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) – input array.
- **x2** (Tensor) – input array.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the element-wise remainder of the quotient `floor_divide(x1, x2)`. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.remainder(np.array([4, 7]), np.array([2, 3]))
>>> print(output)
[0 1]
>>> output = np.remainder(np.arange(7), np.array(5))
>>> print(output)
[0 1 2 3 4 0 1]
```

13.4.99 mindspore.numpy.result_type

mindspore.numpy.**result_type**(**arrays_and_dtypes*)

Returns the type that results from applying the type promotion rules to the arguments.

Note: The promotion rule is slightly different from original Numpy, but more like jax, due to the preference on `32-bit` over `64-bit` data types. Complex dtypes are not supported.

Parameters

***arrays_and_dtypes** (Union[int, float, bool, list, tuple, Tensor, *mindspore.dtype*, str]) –The operands of some operation whose result type is needed.

Returns

mindspore.dtype, the result type.

Raises

TypeError –If the input is not a valid data type.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.result_type('i2', np.float32, True))
Float32
```

13.4.100 mindspore.numpy.rint

mindspore.numpy.**rint**(*x*, *dtype=None*)

Rounds elements of the array to the nearest integer.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (*Union[float, list, tuple, Tensor]*) –Input tensor of any dimension.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Output tensor is same shape and type as x. This is a scalar if x is a scalar.

Raises

TypeError –If *x* can not be converted to tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([-1.7, -1.5, 0.2, 1.5, 1.7, 2.0])
>>> print(np.rint(x))
[-2. -2. 0. 2. 2. 2.]
```

13.4.101 mindspore.numpy.searchsorted

mindspore.numpy.**searchsorted**(*a*, *v*, *side='left'*, *sorter=None*)

Finds indices where elements should be inserted to maintain order. Finds the indices into a sorted array *a* such that, if the corresponding elements in *v* were inserted before the indices, the order of *a* would be preserved.

Parameters

- **a** (*Union[list, tuple, Tensor]*) – 1-D input array. If *sorter* is None, then it must be sorted in ascending order, otherwise *sorter* must be an array of indices that sort it.
- **v** (*Union[int, float, bool, list, tuple, Tensor]*) – Values to insert into *a*.
- **side** (*'left', 'right', optional*) – If `'left'` (default value), the index of the first suitable location found is given. If `'right'` , return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *a*).
- **sorter** (*Union[int, float, bool, list, tuple, Tensor]*) – 1-D optional array of integer indices that sort array *a* into ascending order. They are typically the result of argsort. Default: `None` .

Returns

Tensor, array of insertion points with the same shape as *v*.

Raises

ValueError – If argument for *side* or *sorter* is invalid.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import numpy as np
>>> print(np.searchsorted([1,2,3,4,5], 3))
2
>>> print(np.searchsorted([1,2,3,4,5], 3, side='right'))
3
>>> print(np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3]))
[0 5 1 2]
```

13.4.102 mindspore.numpy.sign

mindspore.numpy.**sign**(*x*, *dtype=None*)

Returns an element-wise indication of the sign of a number.

The sign function returns *-1 if x < 0, 0 if x == 0, 1 if x > 0*. nan is returned for nan inputs.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. Complex inputs are not supported now. On Ascend, integer inputs are not supported.

Parameters

- **x** (*Union[int, float, list, tuple, Tensor]*) –Input values.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

The sign of x. This is a tensor or a scalar when x is a scalar.

Raises

TypeError –If dtype of the input is not in the given types or the input can not be converted to tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> output = np.sign(np.array([-1., 0., 1., 1.2]))
>>> print(output)
[-1.  0.  1.  1.]
```

13.4.103 mindspore.numpy.sin

mindspore.numpy.**sin**(*x*, *dtype=None*)

Trigonometric sine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (`Tensor`) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([-5, -1, 0, 2, 4, 100]).astype('float32')
>>> output = np.sin(x)
>>> print(output)
[ 0.9589243  -0.84147096  0.   0.9092974  -0.7568025  -0.50636566]
```

13.4.104 mindspore.numpy.sinh

mindspore.numpy.**sinh**(*x*, *dtype=None*)

Hyperbolic sine, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5).astype('float32')
>>> print(np.sinh(x))
[ 0.         1.1752012  3.6268604 10.017875  27.289917 ]
```

13.4.105 mindspore.numpy.sqrt

mindspore.numpy.**sqrt**(*x*, *dtype=None*)

Returns the non-negative square-root of an array, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16 and np.float32.

Parameters

- **x** (Tensor) –The values whose square-roots are required.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, an array of the same shape as *x*, containing the positive square-root of each element in *x*. For negative elements, nan is returned. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(6).reshape(2, 3).astype('float32')
>>> x_squared = np.square(x)
>>> output = np.sqrt(x_squared)
>>> print(output)
[[ 0. 1. 2.]
 [ 3. 4. 5.]]
```

13.4.106 mindspore.numpy.square

`mindspore.numpy.`**`square`**(*x*, *dtype=None*)

Returns the element-wise square of the input.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported. On GPU, the supported dtypes are np.float16 and np.float32.

Parameters

- **x** (`Tensor`) – Input data.
- **dtype** (*mindspore.dtype*, optional) – Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, element-wise `x*x`, of the same shape and dtype as *x*. This is a scalar if *x* is a scalar.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.square(np.arange(6).reshape(2, 3).astype('float32'))
>>> print(x)
[[ 0.  1.  4.]
 [ 9. 16. 25.]]
```

13.4.107 mindspore.numpy.std

mindspore.numpy.**std**(*x*, *axis=None*, *ddof=0*, *keepdims=False*)

Computes the standard deviation along the specified axis. The standard deviation is the square root of the average of the squared deviations from the mean, i.e., $std = sqrt(mean(abs(x - x.mean()) **2))$.

Returns the standard deviation, which is computed for the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *dtype*, *out* and *where* are not supported.

Parameters

- **x** (Tensor) −A Tensor to be calculated.
- **axis** (*Union[None, int, tuple(int)]*) −Axis or axes along which the standard deviation is computed. Default: None .

  If None , compute the standard deviation of the flattened array.
- **ddof** (*int*) −Means Delta Degrees of Freedom. The divisor used in calculations is $N - ddof$, where N represents the number of elements. Default: 0.
- **keepdims** −If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor. If the default value is passed, then keepdims will not be passed through to the std method of sub-classes of tensor, however any non-default value will be. If the sub-class' method does not implement keepdims any exceptions will be raised. Default: False .

Returns

Standard deviation tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.array([1., 2., 3., 4.])
>>> output = np.std(input_x)
>>> print(output)
1.118034
```

13.4.108 mindspore.numpy.subtract

mindspore.numpy.**subtract**(*x1*, *x2*, *dtype=None*)

Subtracts arguments, element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (Tensor) −The input to be subtracted from.

- **x2** (Tensor) –The input to be subtracted by.
- **dtype** (*mindspore.dtype*, optional) –Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the difference of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2])
>>> x2 = np.full((3, 2), [3, 4])
>>> output = np.subtract(x1, x2)
>>> print(output)
[[-2 -2]
[-2 -2]
[-2 -2]]
```

13.4.109 mindspore.numpy.sum

mindspore.numpy.**sum**(*a*, *axis=None*, *dtype=None*, *keepdims=False*, *initial=None*)

Returns sum of array elements over a given axis.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **a** (*Union[int, float, bool, list, tuple, Tensor]*) –Elements to sum.
- **axis** (*Union[None, int, tuple(int)], optional*) –Axis or axes along which a sum is performed. Default: None. If *None*, sum all of the elements of the input array. If axis is negative it counts from the last to the first axis. If axis is a tuple of integers, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.
- **dtype** (*mindspore.dtype*, optional) –Defaults to *None*. Overrides the dtype of the output Tensor.
- **keepdims** (*bool, optional*) –If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array. If the default value is passed, then keepdims will not be passed through to the sum method of sub-classes of ndarray, however any non-default value will be. If the sub-class method does not implement keepdims any exceptions will be raised. Default: False.
- **initial** (*scalar, optional*) –Starting value for the sum, if *None*, which refers to the first element of the reduction. Default: None.

Returns

Tensor. An array with the same shape as a, with the specified axis removed. If a is a 0-d array, or if axis is *None*, a scalar is returned. If an output array is specified, a reference to out is returned.

Raises

- **TypeError** –If input is not array_like or *axis* is not int or tuple of integers or *keepdims* is not integer or *initial* is not scalar.
- **ValueError** –If any axis is out of range or duplicate axes exist.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> print(np.sum([0.5, 1.5]))
2.0
>>> x = np.arange(10).reshape(2, 5).astype('float32')
>>> print(np.sum(x, axis=1))
[10. 35.]
```

13.4.110 mindspore.numpy.tan

`mindspore.numpy.`**`tan`**(*x*, *dtype=None*)

Computes tangent element-wise.

Equivalent to $np.sin(x)/np.cos(x)$ element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (`Tensor`) –Input tensor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None`. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Raises

TypeError –If the input is not a tensor or the dtype of tensor is mindspore.float64.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x = np.array([-5, -1, 0, 2, 4, 100]).astype('float32')
>>> print(np.tan(x))
[ 3.380515   -1.5574077   0.         -2.1850398   1.1578213  -0.58721393]
```

13.4.111 mindspore.numpy.tanh

mindspore.numpy.**tanh**(*x*, *dtype=None*)

Computes hyperbolic tangent element-wise.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – Input tensor.
- **dtype** (*mindspore.dtype*, optional) – Default: None. Overrides the dtype of the output Tensor.

Returns

Tensor or scalar. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> x = np.arange(5).astype('float32')
>>> print(np.tanh(x))
[0.        0.7615942 0.9640276 0.9950548 0.9993293]
```

13.4.112 mindspore.numpy.tensordot

mindspore.numpy.**tensordot**(*a*, *b*, *axes=2*)

Computes tensor dot product along specified axes.

Given two tensors, *a* and *b*, and an array_like object containing two array_like objects, *(a_axes, b_axes)*, sum the products of *a*'s and *b*'s elements (components) over the axes specified by *a_axes* and *b_axes*. The third argument can be a single non-negative integer_like scalar, *N*; if it is such, then the last *N* dimensions of *a* and the first *N* dimensions of *b* are summed over. Three common use cases are:

- axes = 0 : tensor product
- axes = 1 : tensor dot product
- axes = 2 : (default) tensor double contraction

When axes is integer_like, the sequence for evaluation will be: first the -*Nth* axis in *a* and 0th axis in *b*, and the -1th axis in *a* and *Nth* axis in *b* last. When there is more than one axis to sum over - and they are not the last (first) axes of *a (b)* - the argument axes should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth. The shape of the result consists of the non-contracted axes of the first tensor, followed by the non-contracted axes of the second.

Note: On CPU, the supported dypes are np.float16 and np.float32. On GPU, the supported dypes are np.float16 and np.float32.

Parameters

- **a** (Tensor) –Tensor to "dot".
- **b** (Tensor) –Tensor to "dot".
- **axes** (*int or sequence of ints*) –integer_like: If an int *N*, sum over the last *N* axes of *a* and the first *N* axes of *b* in order. The sizes of the corresponding axes must match.

  sequence of ints: Or, a list of axes to be summed over, first sequence applying to *a*, second to *b*. Both elements *array_like* must be of the same length.

Returns

Tensor, or list of tensors, the tensor dot product of the input.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.ones((3, 4, 5))
>>> b = np.ones((4, 3, 2))
>>> output = np.tensordot(a, b, axes=([1,0],[0,1]))
>>> print(output.shape)
(5, 2)
```

13.4.113 mindspore.numpy.trapz

mindspore.numpy.**trapz**(*y*, *x=None*, *dx=1.0*, *axis=- 1*)

Integrates along the given axis using the composite trapezoidal rule.

Integrates *y* (x) along given axis.

Parameters

- **y** (Tensor) –Input array to integrate.
- **x** (*Union[int, float, bool, list, tuple, Tensor], optional*) –The sample points corresponding to the *y* values. If *x* is None, the sample points are assumed to be evenly spaced *dx* apart. Default: `None` .
- **dx** (*scalar, optional*) –The spacing between sample points when *x* is None. Default: `1.0` .
- **axis** (*int, optional*) –The axis along which to integrate. Default: `-1` .

Returns

Tensor of float, definite integral as approximated by trapezoidal rule.

Raises

ValueError –If axis is out of range of `[-y.ndim, y.ndim)`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> a = np.arange(6).reshape(2, 3)
>>> output = np.trapz(a,  x=[-2, 1, 2], axis=1)
>>> print(output)
[ 3. 15.]
>>> output = np.trapz(a,  dx=3, axis=0)
>>> print(output)
[ 4.5  7.5 10.5]
```

13.4.114 mindspore.numpy.true_divide

`mindspore.numpy.`**`true_divide`**(*x1*, *x2*, *dtype=None*)

Returns a true division of the inputs, element-wise.

Instead of the Python traditional "floor division", this returns a true division.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x1** (`Tensor`) –the dividend.
- **x2** (`Tensor`) –the divisor.
- **dtype** (*mindspore.dtype*, optional) –Default: `None` . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, this is a scalar if both *x1* and *x2* are scalars.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> x1 = np.full((3, 2), [1, 2])
>>> x2 = np.full((3, 2), [3, 4])
>>> output = np.true_divide(x1, x2)
>>> print(output)
[[0.33333334 0.5       ]
[0.33333334 0.5       ]
[0.33333334 0.5       ]]
```

13.4.115 mindspore.numpy.trunc

mindspore.numpy.**trunc**(*x*, *dtype=None*)

Returns the truncated value of the input, element-wise.

The truncated value of the scalar *x* is the nearest integer *i* which is closer to zero than *x* is. In short, the fractional part of the signed number *x* is discarded.

Note: Numpy arguments *out*, *where*, *casting*, *order*, *subok*, *signature*, and *extobj* are not supported.

Parameters

- **x** (Tensor) – input data.
- **dtype** (*mindspore.dtype*, optional) – Default: None . Overrides the dtype of the output Tensor.

Returns

Tensor or scalar, the truncated value of each element in *x*. This is a scalar if *x* is a scalar.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.numpy as np
>>> output = np.trunc(np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0]))
>>> print(output)
[-1. -1. -0.  0.  1.  1.  2.]
```

13.4.116 mindspore.numpy.unwrap

mindspore.numpy.**unwrap**(*p*, *discont=3.141592653589793*, *axis=- 1*)

Unwraps by changing deltas between values to 2*pi complement. Unwraps radian phase *p* by changing absolute jumps greater than *discont* to their 2*pi complement along the given axis.

Note: For absolute jumps that are within a very close range to pi, unwrapping may be done differently than numpy due to differences in round-off.

Parameters

- **p** (*Union[int, float, bool, list, tuple, Tensor]*) – Input array.
- **discont** (*float, optional*) – Maximum discontinuity between values, default: pi .
- **axis** (*int, optional*) – Axis along which unwrap will operate, default: -1 .

Returns

Tensor.

Raises

ValueError – If the axis is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore.numpy as np
>>> phase = np.add(np.linspace(0, np.pi, num=5), [0, 0, 0, np.pi, np.pi])
>>> print(phase)
[0.        0.7853982 1.5707964 5.4977875 6.2831855]
>>> print(np.unwrap(phase))
[ 0.0000000e+00  7.8539819e-01  1.5707964e+00 -7.8539848e-01 -4.7683716e-07]
```

13.4.117 mindspore.numpy.var

`mindspore.numpy.`**`var`**(*x*, *axis=None*, *ddof=0*, *keepdims=False*)

Computes the variance along the specified axis. The variance is the average of the squared deviations from the mean, i.e., $var = mean(abs(x - x.mean()) **2)$.

Returns the variance, which is computed for the flattened array by default, otherwise over the specified axis.

Note: Numpy arguments *dtype*, *out* and *where* are not supported.

Parameters

- **`x`** (`Tensor`) −A Tensor to be calculated.
- **`axis`** (*`Union[None, int, tuple(int)]`*) −Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array. Default: `None` .
- **`ddof`** (*`int`*) −Means Delta Degrees of Freedom. Default: `0` . The divisor used in calculations is $N - ddof$, where N represents the number of elements.
- **`keepdims`** (*`bool`*) −If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input tensor. If the default value is passed, then keepdims will not be passed through to the var method of sub-classes of tensor, however any non-default value will be. If the sub-class method does not implement keepdims any exceptions will be raised. Default: `False` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Returns

Standard deviation tensor.

Examples

```
>>> import mindspore.numpy as np
>>> input_x = np.array([1., 2., 3., 4.])
>>> output = np.var(input_x)
>>> print(output)
1.25
```

13.5 Interact With MindSpore Functions

Since *mindspore.numpy* directly wraps MindSpore tensors and operators, it has all the advantages and properties of MindSpore. In this section, we will briefly introduce how to employ MindSpore execution management and automatic differentiation in *mindspore.numpy* coding scenarios. These include:

- *jit* decorator: for running codes in static graph mode for better efficiency.
- *GradOperation*: for automatic gradient computation.
- *mindspore.set_context*: for *mindspore.numpy* execution management.
- *mindspore.nn.Cell*: for using *mindspore.numpy* interfaces in MindSpore Deep Learning Models.

The following are examples:

- Use *jit* decorator to run code in static graph mode

  Let's first see an example consisted of matrix multiplication and bias add, which is a typical process in Neural Networks:

```
import mindspore.numpy as np

x = np.arange(8).reshape(2, 4).astype('float32')
w1 = np.ones((4, 8))
b1 = np.zeros((8,))
w2 = np.ones((8, 16))
b2 = np.zeros((16,))
w3 = np.ones((16, 4))
b3 = np.zeros((4,))

def forward(x, w1, b1, w2, b2, w3, b3):
    x = np.dot(x, w1) + b1
    x = np.dot(x, w2) + b2
    x = np.dot(x, w3) + b3
return x

print(forward(x, w1, b1, w2, b2, w3, b3))
```

  The result is as follows:

```
[[ 768.  768.  768.  768.]
 [2816. 2816. 2816. 2816.]]
```

  In this function, MindSpore dispatches each computing kernel to device separately. However, with the help of *jit* decorator, we can compile all operations into a single static computing graph.

```
from mindspore import jit
```

(continues on next page)

(continued from previous page)

```
forward_compiled = jit(forward)
print(forward(x, w1, b1, w2, b2, w3, b3))
```

The result is as follows:

```
[[ 768.  768.  768.  768.]
 [2816. 2816. 2816. 2816.]]
```

Note: Currently, static graph cannot run in Python interactive mode and not all python types can be passed into functions decorated with *jit*.

- Use GradOperation to compute deratives

  GradOperation can be used to take deratives from normal functions and functions decorated with *jit*. Take the previous example:

```
from mindspore import ops

grad_all = ops.GradOperation(get_all=True)
print(grad_all(forward)(x, w1, b1, w2, b2, w3, b3))
```

The result is as follows:

```
(Tensor(shape=[2, 4], dtype=Float32, value=
 [[ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02,  5.12000000e+02],
  [ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02,  5.12000000e+02]]),
 Tensor(shape=[4, 8], dtype=Float32, value=
 [[ 2.56000000e+02,  2.56000000e+02,  2.56000000e+02 ...  2.56000000e+02,  2.56000000e+02, ␣
↪2.56000000e+02],
  [ 3.84000000e+02,  3.84000000e+02,  3.84000000e+02 ...  3.84000000e+02,  3.84000000e+02, ␣
↪3.84000000e+02],
  [ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02 ...  5.12000000e+02,  5.12000000e+02, ␣
↪5.12000000e+02]
  [ 6.40000000e+02,  6.40000000e+02,  6.40000000e+02 ...  6.40000000e+02,  6.40000000e+02, ␣
↪6.40000000e+02]]),
  ...
 Tensor(shape=[4], dtype=Float32, value= [ 2.00000000e+00,  2.00000000e+00,  2.00000000e+00,␣
↪ 2.00000000e+00]))
```

To take the gradient of *jit* compiled functions, first we need to set the execution mode to static graph mode.

```
from mindspore import jit, set_context, GRAPH_MODE, ops

set_context(mode=GRAPH_MODE)
grad_all = ops.GradOperation(get_all=True)
print(grad_all(jit(forward))(x, w1, b1, w2, b2, w3, b3))
```

The result is as follows:

```
(Tensor(shape=[2, 4], dtype=Float32, value=
 [[ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02,  5.12000000e+02],
  [ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02,  5.12000000e+02]]),
 Tensor(shape=[4, 8], dtype=Float32, value=
 [[ 2.56000000e+02,  2.56000000e+02,  2.56000000e+02 ...  2.56000000e+02,  2.56000000e+02, ␣
↪2.56000000e+02],
  [ 3.84000000e+02,  3.84000000e+02,  3.84000000e+02 ...  3.84000000e+02,  3.84000000e+02, ␣
↪3.84000000e+02],
```

(continues on next page)

(continued from previous page)

```
  [ 5.12000000e+02,  5.12000000e+02,  5.12000000e+02 ...  5.12000000e+02,  5.12000000e+02, ␣
↪5.12000000e+02]
  [ 6.40000000e+02,  6.40000000e+02,  6.40000000e+02 ...  6.40000000e+02,  6.40000000e+02, ␣
↪6.40000000e+02]]),
  ...
 Tensor(shape=[4], dtype=Float32, value= [ 2.00000000e+00,  2.00000000e+00,  2.00000000e+00,␣
↪ 2.00000000e+00]))
```

For more details, see API GradOperation .

- Use mindspore.set_context to control execution mode

  Most functions in *mindspore.numpy* can run in Graph Mode and PyNative Mode, and can run on CPU, GPU and Ascend. Like MindSpore, users can manage the execution mode using *mindspore.set_context* :

```
from mindspore import set_context, GRAPH_MODE, PYNATIVE_MODE

# Execution in static graph mode
set_context(mode=GRAPH_MODE)

# Execution in PyNative mode
set_context(mode=PYNATIVE_MODE)

# Execution on CPU backend
set_context(device_target="CPU")

# Execution on GPU backend
set_context(device_target="GPU")

# Execution on Ascend backend
set_context(device_target="Ascend")
...
```

  For more details, see API mindspore.set_context .

- Use mindspore.numpy in MindSpore Deep Learning Models

  mindspore.numpy interfaces can be used inside *nn.cell* blocks as well. For example, the above code can be modified to:

```
import mindspore.numpy as np
from mindspore import set_context, GRAPH_MODE
from mindspore.nn import Cell

set_context(mode=GRAPH_MODE)

x = np.arange(8).reshape(2, 4).astype('float32')
w1 = np.ones((4, 8))
b1 = np.zeros((8,))
w2 = np.ones((8, 16))
b2 = np.zeros((16,))
w3 = np.ones((16, 4))
b3 = np.zeros((4,))

class NeuralNetwork(Cell):
    def construct(self, x, w1, b1, w2, b2, w3, b3):
        x = np.dot(x, w1) + b1
        x = np.dot(x, w2) + b2
        x = np.dot(x, w3) + b3
```

(continues on next page)

(continued from previous page)

```
        return x

net = NeuralNetwork()

print(net(x, w1, b1, w2, b2, w3, b3))
```

The result is as follows:

```
[[ 768.  768.  768.  768.]
 [2816. 2816. 2816. 2816.]]
```

CHAPTER

FOURTEEN

MINDSPORE.SCIPY

Warning: These are experimental APIs that are subject to change or deletion. Only support Linux.

Scipy-like interfaces in mindspore.

14.1 mindspore.scipy.linalg

Linear algebra submodule

API Name	Description	Supported Platforms
mindspore.scipy.linalg.block_diag	Create a block diagonal matrix from provided arrays.	GPU CPU
mindspore.scipy.linalg.cho_factor	Compute the cholesky decomposition of a matrix, to use in *mindspore.scipy.linalg.cho_solve()*.	GPU CPU
mindspore.scipy.linalg.cho_solve	Given the cholesky factorization of A, solve the linear equation.	GPU CPU
mindspore.scipy.linalg.cholesky	Compute the cholesky decomposition of a matrix.	GPU CPU
mindspore.scipy.linalg.eigh	Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.	GPU CPU
mindspore.scipy.linalg.inv	Compute the inverse of a matrix.	GPU CPU
mindspore.scipy.linalg.lstsq	Computes a solution to the least squares problem of a system of linear equations $AX = B$.	Ascend CPU
mindspore.scipy.linalg.lu	Compute pivoted LU decomposition of a general matrix.	GPU CPU
mindspore.scipy.linalg.lu_factor	Compute pivoted LU decomposition of a square matrix, and its outputs can be directly used as the inputs of *lu_solve*.	GPU CPU
mindspore.scipy.linalg.solve_triangular	Solve the linear system $ax = b$ for x, Assuming a is a triangular matrix.	Ascend CPU

14.1.1 mindspore.scipy.linalg.block_diag

mindspore.scipy.linalg.**block_diag**(**arrs*)

Create a block diagonal matrix from provided arrays.

Given the list of Tensors *A*, *B*, and *C*, the output will have these Tensors arranged on the diagonal:

```
[[A, 0, 0],
 [0, B, 0],
 [0, 0, C]]
```

Note: *block_diag* is not supported on Windows platform yet.

Parameters

arrs (*list*) – up to 2-D Input Tensors. One or more Tensors, the dimension of Tensors should be 0-D, 1-D or 2-D.

Returns

Tensor with *A*, *B*, *C*, ··· on the diagonal which has the same dtype as *A*.

Raises

ValueError – If there are Tensors with dimensions higher than 2 in all arguments.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import block_diag
>>> A = Tensor(onp.array([[1, 0], [0, 1]]))
>>> B = Tensor(onp.array([[3, 4, 5], [6, 7, 8]]))
>>> C = Tensor(onp.array([[7]]))
>>> P = Tensor(onp.zeros((2, ), dtype='int32'))
>>> print(block_diag(A, B, C))
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 3 4 5 0]
 [0 0 6 7 8 0]
 [0 0 0 0 0 7]]
>>> print(block_diag(A, P, B, C))
[[1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 3 4 5 0]
 [0 0 0 0 6 7 8 0]
 [0 0 0 0 0 0 0 7]]
```

14.1.2 mindspore.scipy.linalg.cho_factor

mindspore.scipy.linalg.**cho_factor**(*a*, *lower=False*, *overwrite_a=False*, *check_finite=True*)

Compute the cholesky decomposition of a matrix, to use in *mindspore.scipy.linalg.cho_solve()*.

Returns the cholesky decomposition of a Hermitian positive-definite matrix A. Base on the value of *lower*, perform the following decomposition:

- when *lower* is True: $A = LL^*$
- when *lower* is False: $A = U^*U$

L^* is a conjugate transpose matrix of L. U^* is a conjugate transpose matrix of U.

The return value can be directly used as the first parameter to *mindspore.scipy.linalg.cho_solve()*.

Note:

- *cho_factor* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Warning: The returned matrix also contains random data in the entries not used by the cholesky decomposition. If you need to zero these entries, use the function *cholesky* instead.

Parameters

- **a** (Tensor) −square Matrix of (M, M) to be decomposed.
- **lower** (*bool, optional*) −Whether to compute the upper or lower triangular cholesky factorization. Default: `False`.
- **overwrite_a** (*bool, optional*) −Whether to overwrite data in a (may improve performance). Default: `False`. in mindspore, this arg does not work right now.
- **check_finite** (*bool, optional*) −Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: `True`. in mindspore, this arg does not work right now.

Returns

- Tensor, matrix whose upper or lower triangle contains the cholesky factor of a. Other parts of the matrix contain random data.
- bool, flag indicating whether the factor is in the lower or upper triangle

Raises

ValueError −If input a tensor is not a square matrix or it's dims not equal to 2D.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import cho_factor
>>> a = Tensor(onp.array([[9, 3, 1, 5], [3, 7, 5, 1], [1, 5, 9, 2], [5, 1, 2, 6]]).
↪astype(onp.float32))
>>> c, low = cho_factor(a)
>>> print(c)
[[ 3.          1.          0.33333334  1.6666666 ]
 [ 3.          2.4494898   1.9051585  -0.2721655 ]
 [ 1.          5.          2.2933078   0.8559526 ]
 [ 5.          1.          2.          1.5541857 ]]
```

14.1.3 mindspore.scipy.linalg.cho_solve

mindspore.scipy.linalg.**cho_solve**(*c_and_lower*, *b*, *overwrite_b=False*, *check_finite=True*)

Given the cholesky factorization of A, solve the linear equation.

$$Ax = b$$

Note:

- *cho_solve* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **c_and_lower** (*(Tensor, bool)*) – cholesky factorization of a, as given by *mindspore.scipy.linalg.cho_factor()*.
- **b** (Tensor) – Right-hand side.
- **overwrite_b** (*bool, optional*) – Whether to overwrite data in b (may improve performance). Default: `False`.
- **check_finite** (*bool, optional*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: `True`.

Returns

Tensor, the solution to the system $Ax = b$.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> import mindspore as ms
>>> a = ms.Tensor(onp.array([[9, 3, 1, 5], [3, 7, 5, 1], [1, 5, 9, 2], [5, 1, 2, 6]]).
↪astype(onp.float32))
>>> b = ms.Tensor(onp.array([1, 1, 1, 1]).astype(onp.float32))
>>> c, low = ms.scipy.linalg.cho_factor(a)
>>> x = ms.scipy.linalg.cho_solve((c, low), b)
>>> print(x)
[-0.01749266  0.11953348  0.01166185  0.15743434]
```

14.1.4 mindspore.scipy.linalg.cholesky

mindspore.scipy.linalg.**cholesky**(*a*, *lower=False*, *overwrite_a=False*, *check_finite=True*)

Compute the cholesky decomposition of a matrix.

Returns the cholesky decomposition of a Hermitian positive-definite matrix A. Base on the value of *lower*, perform the following decomposition:

- when *lower* is True: $A = LL^*$
- when *lower* is False: $A = U^*U$

L^* is a conjugate transpose matrix of L. U^* is a conjugate transpose matrix of U.

Note:

- *cholesky* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **a** (Tensor) –square Matrix of (M, M) to be decomposed.
- **lower** (*bool, optional*) –Whether to compute the upper- or lower-triangular cholesky factorization. Default: False.
- **overwrite_a** (*bool, optional*) –Whether to overwrite data in *a* (may improve performance). Default: False . in mindspore, this arg does not work right now.
- **check_finite** (*bool, optional*) –Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True . in mindspore, this arg does not work right now.

Returns

Tensor, upper- or lower-triangular cholesky factor of *a*.

Raises

ValueError –If input a tensor is not a square matrix or it's dims not equal to 2D.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import cholesky
>>> a = Tensor(onp.array([[1, 2],[2, 5]]).astype(onp.float32))
>>> L = cholesky(a, lower=True)
>>> print(L)
[[1. 0.]
 [2. 1.]]
```

14.1.5 mindspore.scipy.linalg.eigh

mindspore.scipy.linalg.**eigh**(*a*, *b=None*, *lower=True*, *eigvals_only=False*, *overwrite_a=False*, *overwrite_b=False*, *turbo=True*, *eigvals=None*, *type=1*, *check_finite=True*)

Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues Tensor *w* and optionally eigenvectors Tensor *v* of Tensor *a*, where *b* is positive definite such that for every eigenvalue λ (i-th entry of w) and its eigenvector *vi* (i-th column of *v*) satisfies:

```
                a @ vi = λ * b @ vi
vi.conj().T @ a @ vi = λ
vi.conj().T @ b @ vi = 1
```

In the standard problem, *b* is assumed to be the identity matrix.

Note:

- *eigh* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **a** (Tensor) – A (M, M) complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.
- **b** (Tensor, *optional*) – A (M, M) complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed. Default: None.
- **lower** (*bool*, *optional*) – Whether the pertinent Tensor data is taken from the lower or upper triangle of *a* and, if applicable, *b*. Default: True.
- **eigvals_only** (*bool*, *optional*) – Whether to calculate only eigenvalues and no eigenvectors. Default: False.
- **overwrite_a** (*bool*, *optional*) – Whether to overwrite data in *a* (may improve performance). Default: False.
- **overwrite_b** (*bool*, *optional*) – Whether to overwrite data in *b* (may improve performance). Default: False.
- **turbo** (*bool*, *optional*) – use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if full set of eigenvalues are requested.). Has no significant effect if eigenvectors are not requested. Default: True .

- **eigvals** (*tuple, optional*) –Indexes of the smallest and largest (in ascending order) eigenvalues and corresponding eigenvectors to be returned: $0 <= lo <= hi <= M - 1$. If omitted, all eigenvalues and eigenvectors are returned. Default: `None` .
- **type** (*int, optional*) –For the generalized problems, this keyword specifies the problem type to be solved for w and v (only takes 1, 2, 3 as possible inputs):

```
1 =>     a @ v = w @ b @ v
2 => a @ b @ v = w @ v
3 => b @ a @ v = w @ v
```

  This keyword is ignored for standard problems. Default: `1` .
- **check_finite** (*bool, optional*) –Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: `True` .

Returns

- Tensor with shape $(N,)$, the $N(1 <= N <= M)$ selected eigenvalues, in ascending order, each repeated according to its multiplicity.
- Tensor with shape (M, N), if *eigvals_only* == *False*.

Raises

- **RuntimeError** –If eigenvalue computation does not converge, an error occurred, or b matrix is not definite positive. Note that if input matrices are not symmetric or Hermitian, no error will be reported but results will be wrong.
- **TypeError** –If *a* is not Tensor.
- **TypeError** –If *lower* is not bool.
- **TypeError** –If *eigvals_only* is not bool.
- **TypeError** –If *overwrite_a* is not bool.
- **TypeError** –If *overwrite_b* is not bool.
- **TypeError** –If *turbo* is not bool.
- **TypeError** –If *check_finite* is not bool.
- **ValueError** –If *a* is not 2D square matrix.
- **ValueError** –If *b* is not None.
- **ValueError** –If *eigvals* is not None.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> import mindspore.numpy as mnp
>>> from mindspore import Tensor, dtype
>>> from mindspore.scipy.linalg import eigh
>>> a = Tensor([[6, 3, 1, 5], [3, 0, 5, 1], [1, 5, 6, 2], [5, 1, 2, 2]], dtype.float64)
>>> w, v = eigh(a)
>>> print(onp.allclose(mnp.dot(a, v).asnumpy(), mnp.dot(v, mnp.diag(w)).asnumpy(), 1e-5, 1e-
↪8))
True
```

14.1.6 mindspore.scipy.linalg.inv

mindspore.scipy.linalg.**inv**(*a*, *overwrite_a=False*, *check_finite=True*)

Compute the inverse of a matrix.

Note:

- *inv* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **a** (Tensor) –Square matrix to be inverted.
- **overwrite_a** (*bool, optional*) –Discard data in *a* (may improve performance). Default: False .
- **check_finite** (*bool, optional*) –Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True .

Returns

Tensor, inverse of the matrix *a*.

Raises

- **LinAlgError** –If *a* is singular.
- **ValueError** –If *a* is not square, or not 2D.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> import mindspore.numpy as mnp
>>> from mindspore.scipy.linalg import inv
>>> a = Tensor(onp.array([[1., 2.], [3., 4.]]))
>>> print(inv(a))
[[-2.   1. ]
 [ 1.5 -0.5]]
>>> print(mnp.dot(a, inv(a)))
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

14.1.7 mindspore.scipy.linalg.lstsq

mindspore.scipy.linalg.**lstsq**(*A*, *B*, *rcond=None*, *driver=None*)

Computes a solution to the least squares problem of a system of linear equations $AX = B$.

Note:

- *lstsq* is currently only used in *mindscience* scientific computing scenarios and dose not support other usage scenarios.
- *lstsq* is not supported on Windows platform yet.

Parameters

- **A** (Tensor) –LHS input tensor of shape $(*, M, N)$, where $*$ is zero or more batch dimensions.
- **B** (Tensor) –RHS input tensor of shape $(*, M, K)$, where $*$ is zero or more batch dimensions.
- **rcond** (*number.Number, optional*) –Not implemented now, Default is None.
- **driver** (*string, optional*) –Which LAPACK driver is used to solve the least-squares problem. Options are "gels", "gelsy", "gelss", "gelsd". Default is None ("gelsy"). if A is well-conditioned, "gels" is a good choice for full-rank matrix, and "gelsy" for a general matrix. if A is not well-conditioned, "gelsd" works good, "gelss" was used historically. It is generally slow but uses less memory.

Returns

- **solution** (Tensor), Least-squares solution. It has shape $(*, N, K)$, where $*$ is same as broadcast batch dimensions.
- **residual** (Tensor), Square of the 2-norm for each column in $AX - B$, It has shape $(*, K)$, where $*$ is same as broadcast batch dimensions. It is computed when *driver* is one of ("gels", "gelss", "gelsd") and $M > N$, otherwise, it is an empty tensor.
- **rank** (Tensor), Effective rank of A. It has shape $(*)$, where $*$ is same as batch dimensions of A. It is computed when *driver* is one of ("gelsy", "gelss", "gelsd"), otherwise it is an empty tensor.
- **singular_value** (Tensor), Singular values of A. It has shape $(*, min(M, N))$, where $*$ is same as batch dimensions of A. It is computed when *driver* is one of ("gelss", "gelsd"), otherwise it is an empty tensor.

Raises

- **TypeError** –If dtype of A and B are not the same.
- **ValueError** –If A is less than 2 dimension.
- **ValueError** –If the shape of A and B are not matched.

- **ValueError** –If *driver* is not in set {None, "gels", "gelsy", "gelss", "gelsd"}.

Supported Platforms:

Ascend CPU

Examples

```
>>> import numpy as onp
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import lstsq
>>> a = Tensor(onp.array([[3, 0, 0, 0], [2, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 1]], onp.
↪float32))
>>> b = Tensor(onp.array([3, 1, 3, 4], onp.float32))
>>> solution, residual, rank, singular_value = lstsq(a, b)
>>> print(solution)
[ 1. -1.  2.  2.]
>>> print(a @ solution)  # Check the result
[3. 1. 3. 4.]
```

14.1.8 mindspore.scipy.linalg.lu

mindspore.scipy.linalg.**lu**(*a*, *permute_l=False*, *overwrite_a=False*, *check_finite=True*)

Compute pivoted LU decomposition of a general matrix.

The decomposition is:

$$A = PLU$$

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Note:

- *lu* is not supported on Windows platform yet.
- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **a** (Tensor) –a (M, N) matrix to decompose. Note that if the input tensor is not a float, then it will be cast to mstype.float32.
- **permute_l** (*bool, optional*) –Perform the multiplication PL (Default: do not permute). Default: False .
- **overwrite_a** (*bool, optional*) –Whether to overwrite data in *a* (may improve performance). Default: False .
- **check_finite** (*bool, optional*) –Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: True .

Returns

If permute_l == False

- Tensor, (M, M) permutation matrix.
- Tensor, (M, K) lower triangular or trapezoidal matrix with unit diagonal. $K = min(M, N)$.
- Tensor, (K, N) upper triangular or trapezoidal matrix.

If permute_l == True

- Tensor, (M, K) permuted L matrix. $K = min(M, N)$.
- Tensor, (K, N) upper triangular or trapezoidal matrix.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import lu
>>> a = Tensor(onp.array([[2, 5, 8, 7], [5, 2, 2, 8], [7, 5, 6, 6], [5, 4, 4, 8]]).
↪astype(onp.float64))
>>> p, l, u = lu(a)
>>> print(p)
[[0 1 0 0]
 [0 0 0 1]
 [1 0 0 0]
 [0 0 1 0]]
>>> print(l)
[[ 1.          0.          0.          0.        ]
 [ 0.2857143   1.          0.          0.        ]
 [ 0.71428573  0.12        1.          0.        ]
 [ 0.71428573 -0.44       -0.46153846  1.        ]]
>>> print(u)
[[ 7.          5.          6.          6.        ]
 [ 0.          3.57142854  6.28571415  5.28571415]
 [ 0.          0.         -1.03999996  3.07999992]
 [ 0.         -0.         -0.          7.46153831]]
```

14.1.9 mindspore.scipy.linalg.lu_factor

mindspore.scipy.linalg.**lu_factor**(*a*, *overwrite_a=False*, *check_finite=True*)

Compute pivoted LU decomposition of a square matrix, and its outputs can be directly used as the inputs of *lu_solve*. The decomposition is:

$$a = PLU$$

where P is a permutation matrix, L lower triangular with unit diagonal elements, and U upper triangular.

Note:

- *lu_factor* is not supported on Windows platform yet.

- Only float32, float64, int32, int64 are supported Tensor dtypes.
- If Tensor with dtype int32 or int64 is passed, it will be cast to mstype.float64.

Parameters

- **a** (`Tensor`) –square matrix of (M, M) to decompose. Note that if the input tensor is not a float, then it will be cast to mstype.float32.
- **overwrite_a** (*bool, optional*) –Whether to overwrite data in a (may increase performance). Default: `False` .
- **check_finite** (*bool, optional*) –Whether to check that the input matrix contains only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs. Default: `True` .

Returns

- Tensor, a square matrix of (M, M) containing U in its upper triangle, and L in its lower triangle. The unit diagonal elements of L are not stored.
- Tensor, $(M,)$ pivot indices representing the permutation matrix P: the i-th element value j in the indices indicates that row i of matrix was interchanged with row j.

Raises

ValueError –If a is not 2D square.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import lu_factor
>>> a = Tensor(onp.array([[2, 5, 8, 7], [5, 2, 2, 8], [7, 5, 6, 6], [5, 4, 4, 8]]).
↪astype(onp.float64))
>>> lu, piv = lu_factor(a)
>>> print(lu)
[[ 7.          5.          6.          6.        ]
 [ 0.28571429  3.57142857  6.28571429  5.28571429]
 [ 0.71428571  0.12       -1.04        3.08      ]
 [ 0.71428571 -0.44       -0.46153846  7.46153846]]
>>> print(piv)
[2 2 3 3]
```

14.1.10 mindspore.scipy.linalg.solve_triangular

mindspore.scipy.linalg.**solve_triangular**(*a*, *b*, *trans=0*, *lower=False*, *unit_diagonal=False*, *overwrite_b=False*, *debug=None*, *check_finite=True*)

Solve the linear system $ax = b$ for x, Assuming a is a triangular matrix.

Note:

- *solve_triangular* is currently only used in *mindscience* scientific computing scenarios and dose not support other usage scenarios.
- *solve_triangular* is not supported on Windows platform yet.

Parameters

- **a** (Tensor) −A triangular matrix of shape $(*, M, M)$ where $*$ is zero or more batch dimensions.
- **b** (Tensor) −A Tensor of shape $(*, M)$ or $(*, M, N)$. Right-hand side matrix in $ax = b$.
- **trans** (*Union[int, str], optional*) −Type of system to solve. Default: 0.

trans	system
0 or 'N'	a x = b
1 or 'T'	a^T x = b
2 or 'C'	a^H x = b

- **lower** (*bool, optional*) −Use only data contained in the lower triangle of a. Default: False.
- **unit_diagonal** (*bool, optional*) −If True, diagonal elements of a are assumed to be 1 and will not be referenced. Default: False.
- **overwrite_b** (*bool, optional*) −Not implemented now. Default: False.
- **debug** (*Any, optional*) −Not implemented now. Default: None.
- **check_finite** (*bool, optional*) −Not implemented now. Default: True.

Returns

Tensor of shape $(*, M)$ or $(*, M, N)$, which is the solution to the system $ax = b$. Shape of x matches b.

Raises

- **ValueError** −If a is less than 2 dimension.
- **ValueError** −if a is not square matrix.
- **TypeError** −If dtype of a and b are not the same.
- **ValueError** −If the shape of a and b are not matched.
- **ValueError** −If *trans* is not in set {0, 1, 2, 'N', 'T', 'C'}.

Supported Platforms:

Ascend CPU

Examples

```
>>> import numpy as onp
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.scipy.linalg import solve_triangular
>>> a = Tensor(onp.array([[3, 0, 0, 0], [2, 1, 0, 0], [1, 0, 1, 0], [1, 1, 1, 1]], onp.
↪float32))
>>> b = Tensor(onp.array([3, 1, 3, 4], onp.float32))
>>> x = solve_triangular(a, b, lower=True, unit_diagonal=False, trans='N')
>>> print(x)
[ 1. -1.  2.  2.]
>>> print(a @ x)  # Check the result
[3. 1. 3. 4.]
```

14.2 mindspore.scipy.optimize

Optimize submodule

API Name	Description	Supported Platforms
mindspore.scipy.optimize.line_search	Inexact line search that satisfies strong Wolfe conditions.	GPU CPU
mindspore.scipy.optimize.linear_sum_assignment	Solve the linear sum assignment problem.	Ascend CPU
mindspore.scipy.optimize.minimize	Minimization of scalar function of one or more variables.	GPU CPU

14.2.1 mindspore.scipy.optimize.line_search

mindspore.scipy.optimize.**line_search**(*f*, *xk*, *pk*, *jac=None*, *gfk=None*, *old_fval=None*, *old_old_fval=None*, *c1=1e-4*, *c2=0.9*, *maxiter=20*)

Inexact line search that satisfies strong Wolfe conditions.

Algorithm 3.5 from Wright and Nocedal, 'Numerical Optimization', 1999, pg. 59-61

Note: *line_search* is not supported on Windows platform yet.

Parameters

- **f** (*function*) –function of the form f(x) where x is a flat Tensor and returns a real scalar. The function should be composed of operations with vjp defined.
- **xk** (Tensor) –initial guess.
- **pk** (Tensor) –direction to search in. Assumes the direction is a descent direction.
- **jac** (*function*) –the gradient function at x where x is a flat Tensor and returns a Tensor. The function can be None if you want to use automatic credits.
- **gfk** (Tensor) –initial value of value_and_gradient as position. Default: None .
- **old_fval** (Tensor) –The same as *gfk*. Default: None .

- **old_old_fval** (Tensor) −unused argument, only for scipy API compliance. Default: None .
- **c1** (*float*) −Wolfe criteria constant, see ref. Default: 1e-4 .
- **c2** (*float*) −The same as *c1*. Default: 0.9 .
- **maxiter** (*int*) −maximum number of iterations to search. Default: 20 .

Returns

LineSearchResults, results of line search results.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore.scipy.optimize import line_search
>>> from mindspore import Tensor
>>> x0 = Tensor(onp.ones(2).astype(onp.float32))
>>> p0 = Tensor(onp.array([-1, -1]).astype(onp.float32))
>>> def func(x):
...     return x[0] ** 2 - x[1] ** 3
>>> res = line_search(func, x0, p0)
>>> print(res.a_k)
1.0
```

14.2.2 mindspore.scipy.optimize.linear_sum_assignment

mindspore.scipy.optimize.**linear_sum_assignment**(*cost_matrix*, *maximize*, *dimension_limit=Tensor(sys.maxsize)*)

Solve the linear sum assignment problem.

The assignment problem is represented as follows:

$$min \sum_i \sum_j C_{i,j} X_{i,j}$$

where C is cost matrix, $X_{i,j} = 1$ means column j is assigned to row i .

Parameters

- **cost_matrix** (Tensor) −2-D cost matrix. Tensor of shape (M, N) .
- **maximize** (*bool*) −Calculate a maximum weight matching if true, otherwise calculate a minimum weight matching.
- **dimension_limit** (Tensor, *optional*) −A scalar used to limit the actual size of the 2nd dimension of cost_matrix. Default is Tensor(sys.maxsize), which means no limitation. The type is 0-D int64 Tensor.

Returns

A tuple of tensors containing 'row_idx' and 'col_idx'.

- **row_idx** (Tensor) - Row indices of the problem. If *dimension_limit* is given, -1 would be padded at the end. The shape is $(N,)$, where N is the minimum value of *cost_matrix* dimension.
- **col_idx** (Tensor) - Column indices of the problem. If *dimension_limit* is given, -1 would be padded at the end. The shape is $(N,)$, where N is the minimum value of *cost_matrix* dimension.

Raises

- **TypeError** – If the data type of *cost_matrix* is not the type in [float16, float32, float64, int8, int16, int32, int64, uint8, uint16, uint32, uint64, bool]
- **TypeError** – If the type of *maximize* is not bool.
- **TypeError** – If the data type of *dimension_limit* is not int64.
- **ValueError** – If the rank of *cost_matrix* is not 2.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor
>>> import mindspore.scipy.optimize.linear_sum_assignment as lsap
>>> cost_matrix = Tensor(np.array([[2, 3, 3], [3, 2, 3], [3, 3, 2]])).astype(ms.float64)
>>> dimension_limit = Tensor(2)
>>> maximize = False
>>> a, b = lsap(cost_matrix, maximize, dimension_limit)
>>> print(a)
[0 1 -1]
>>> print(b)
[0 1 -1]
>>> a, b = lsap(cost_matrix, maximize)
>>> print(a)
[0 1 2]
>>> print(b)
[0 1 2]
```

14.2.3 mindspore.scipy.optimize.minimize

mindspore.scipy.optimize.**minimize**(*func*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

Minimization of scalar function of one or more variables.

This API for this function matches SciPy with some minor deviations:

- Gradients of `func` are calculated automatically using MindSpore's autodiff support when the value of jac is None.
- The `method` argument is required. A exception will be thrown if you don't specify a solver.
- Various optional arguments *"hess"*, *"hessp"*, *"bounds"*, *"constraints"*, *"tol"*, *"callback"* in the SciPy interface have not yet been implemented.
- Optimization results may differ from SciPy due to differences in the line search implementation.

Note:

- *minimize* does not yet support differentiation or arguments in the form of multi-dimensional Tensor, but support for both is planned.
- *minimize* is not supported on Windows platform yet.

- *LAGRANGE* method is only supported on "GPU".

Parameters

- **func** (*Callable*) −the objective function to be minimized, $fun(x, *args) -> float$, where x is a 1-D array with shape $(n,)$ and *args* is a tuple of the fixed parameters needed to completely specify the function. *fun* must support differentiation if jac is None.
- **x0** (Tensor) −initial guess. Array of real elements of size $(n,)$, where n is the number of independent variables.
- **args** (*Tuple*) −extra arguments passed to the objective function. Default: `()` .
- **method** (*str*) −solver type. Should be one of *"BFGS"* and *"LBFGS"*, *"LAGRANGE"*.
- **jac** (*Callable, optional*) −method for computing the gradient vector. Only for *"BFGS"* and *"LBFGS"*. if it is None, the gradient will be estimated with gradient of `func`. if it is a callable, it should be a function that returns the gradient vector: $jac(x, *args) -> array\_like, shape(n,)$ where x is an array with shape $(n,)$ and args is a tuple with the fixed parameters.
- **hess** (*Callable, optional*) −Method for calculating the Hessian Matrix. Not implemented yet.
- **hessp** (*Callable, optional*) −Hessian of objective function times an arbitrary vector p. Not implemented yet.
- **bounds** (*Sequence, optional*) −Sequence of *(min, max)* pairs for each element in x. Not implemented yet.
- **constraints** (*Callable, optional*) −representing the inequality constrains, each function in constrains indicates the function < 0 as an inequality constrain.
- **tol** (*float, optional*) −tolerance for termination. For detailed control, use solver-specific options. Default: `None` .
- **callback** (*Callable, optional*) −A callable called after each iteration. Not implemented yet.
- **options** (*Mapping[str, Any], optional*) −a dictionary of solver options. All methods accept the following generic options. Default: `None` .
  - history_size (int): size of buffer used to help to update inv hessian, only used with method="LBFGS". Default: `20` .
  - maxiter (int): Maximum number of iterations to perform. Depending on the method each iteration may use several function evaluations.

  The follow options are exclusive to Lagrange method:
  - save_tol (list): list of saving tolerance, with the same length with 'constrains'.
  - obj_weight (float): weight for objective function, usually between 1.0 - 100000.0.
  - lower (Tensor): lower bound constrain for variables, must have same shape with x0.
  - upper (Tensor): upper bound constrain for variables, must have same shape with x0.
  - learning_rate (float): learning rate for each Adam step.
  - coincide_func (Callable): sub-function representing the common parts between objective function and constrains to avoid redundant computation.
  - rounds (int): times to update Lagrange multipliers.
  - steps (int): steps to apply Adam per round.
  - log_sw (bool): whether to print the loss at each step.

Returns

OptimizeResults, object holding optimization results.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as onp
>>> from mindspore.scipy.optimize import minimize
>>> from mindspore import Tensor
>>> x0 = Tensor(onp.zeros(2).astype(onp.float32))
>>> def func(p):
...     x, y = p
...     return (x ** 2 + y - 11.) ** 2 + (x + y ** 2 - 7.) ** 2
>>> res = minimize(func, x0, method='BFGS', options=dict(maxiter=None, gtol=1e-6))
>>> print(res.x)
[3. 2.]
>>> l_res = minimize(func, x0, method='LBFGS', options=dict(maxiter=None, gtol=1e-6))
>>> print(l_res.x)
[3. 2.]
```

CHAPTER

FIFTEEN

MINDSPORE.MULTIPROCESSING

15.1 Overview

mindspore.multiprocessing provides the ability to create multiple processes. Its internal implementation inherits from Python native multiprocessing module and overloads some of the interfaces to ensure that the MindSpore framework works properly when creating multiple processes by fork.

When using mindspore.multiprocessing and creating multiprocesses by fork, the framework internally cleans up and resets threads, locks, and other resources to ensure that the framework functions properly.

- When fork occurs, the parent process waits for the internal thread task to finish executing before the fork, and actively holds the GIL lock in the current thread to avoid the child process after the fork from being unable to hold the GIL lock.
- After fork occurs, the parent process releases the actively held GIL lock.
- After fork occurs, the child process releases the actively held GIL lock, then restores and cleans up resources such as threads within the framework, and resets the backend to `CPU`.

The above process is not triggered when creating a multiprocess without using fork. At this point, the process of creating a multiprocess using mindspore.multiprocessing is exactly the same as that of using Python native multiprocessing module.

Since mindspore.multiprocessing inherits from Python native multiprocessing module, its interface usage is fully compatible with the native module, so users can directly change `import multiprocessing` to `import mindspore.multiprocessing`. Therefore, the description of the native interfaces is not repeated here. For a detailed description and the usage description of the interface, please refer to multiprocessing.

> Only Tensors with `CPU` backend can be shared between processes now.

15.2 Usage Description

A sample of creating multiple processes using mindspore.multiprocessing is shown below:

```
from mindspore import Tensor, ops
import mindspore.multiprocessing as multiprocessing

def child_process():
    y = ops.log(Tensor(2.0))
    print("ops.log(Tensor(2.0))=", y)

if __name__ == '__main__':
    p = multiprocessing.Process(target=child_process)
    p.start()
    p.join()
```

The result is shown below:

```
ops.log(Tensor(2.0))= 0.6931472
```

A sample of creating a process pool using mindspore.multiprocessing is shown below:

```
from mindspore import Tensor, ops
import mindspore.multiprocessing as multiprocessing

def child_process(x):
    return ops.log(x)

if __name__ == '__main__':
    with multiprocessing.Pool(processes=2) as pool:
        inputs = [Tensor(2.0), Tensor(2.0)]
        outputs = pool.map(child_process, inputs)
        print("ops.log(Tensor(2.0))=", outputs)
```

The result is shown below:

```
ops.log(Tensor(2.0))= [Tensor(shape=[], dtype=Float32, value= 0.693147), Tensor(shape=[],␣
↪dtype=Float32, value= 0.693147)]
```

When the user creates a multiprocess by fork after executing a computing task, if the module used is not mindspore.multiprocessing, there may be a problem that the child process is stuck due to the loss of the frame thread. Changing to use the fork of the mindspore.multiprocessing module to create multiprocesses can solve this problem.

The fork of creating child processes is only supported on POSIX systems (e.g. Linux and macOS), but not on Windows.

```
from mindspore import Tensor, ops
# Child process may be stuck when using 'import multiprocessing' .
import mindspore.multiprocessing as multiprocessing

def child_process(q):
    y = ops.log(Tensor(2.0))
    q.put(y)
    return

if __name__ == '__main__':
    multiprocessing.set_start_method("fork", force=True)
    print("parent process:ops.log(Tensor(2.0))=", ops.log(Tensor(2.0)))
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=child_process, args=(q,))
    p.start()
    print("child process:ops.log(Tensor(2.0))=", q.get())
    p.join()
```

The result is shown below:

```
parent process:ops.log(Tensor(2.0))= 0.6931472
child process:ops.log(Tensor(2.0))= 0.6931472
```

When the backend is `Ascend`, the process has exclusive access to the card resources. If a user creates a child process after performing a computation task, and the child process uses the resources of the parent process to perform another computation task, it may report an error due to resource inaccessibility. After modifying the child process creation method to fork, the framework will reset the backend of the child process to `CPU` to avoid resource conflicts.

```
from mindspore import Tensor, ops, context
import mindspore.multiprocessing as multiprocessing

def child_process(q):
    y = ops.log(Tensor(2.0))
    q.put(y)
    return

if __name__ == '__main__':
    context.set_context(device_target="Ascend")
    # Child process may not be able to acquire resources when start_method is set to 'spawn' .
    multiprocessing.set_start_method("fork", force=True)
    print("parent process:ops.log(Tensor(2.0))=", ops.log(Tensor(2.0)))
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=child_process, args=(q,))
    p.start()
    print("child process:ops.log(Tensor(2.0))=", q.get())
    p.join()
```

The result is shown below:

```
parent process:ops.log(Tensor(2.0))= 0.6931472
child process:ops.log(Tensor(2.0))= 0.6931472
```

The action of creating a multiprocess can be placed in front of executing any computational tasks an the backend is modified manually in a child process.

```
from mindspore import Tensor, ops, context
import mindspore.multiprocessing as multiprocessing

def child_process(q):
    # Child process may not be able to acquire resources when using the same resources as parent␣
↪process.
    context.set_context(device_target="CPU")
    y = ops.log(Tensor(2.0))
    q.put(y)
    return

if __name__ == '__main__':
    context.set_context(device_target="Ascend")
    multiprocessing.set_start_method("spawn", force=True)
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=child_process, args=(q,))
    p.start()
    print("child process:ops.log(Tensor(2.0))=", q.get())
    p.join()
    # Child process may not be able to acquire resources when this compute task is executed␣
↪before the child process is created.
    print("parent process:ops.log(Tensor(2.0))=", ops.log(Tensor(2.0)))
```

The result is shown below:

```
child process:ops.log(Tensor(2.0))= 0.6931472
parent process:ops.log(Tensor(2.0))= 0.6931472
```

CHAPTER

SIXTEEN

MINDSPORE.UTILS

mindspore.utils.**stress_detect**()

Inspect the hardware to determine if there are any faults affecting its accuracy and precision.Common use cases include invoking this interface at each step or when saving checkpoints, allowing users to check if any hardware issues could impact precision.

Returns

int, the return value represents the error type: zero indicates normal operation; non-zero values indicate a hardware failure.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore.utils import stress_detect
>>> ret = stress_detect()
>>> print(ret)
0
```

mindspore.utils.dryrun.**set_simulation**()

This interface is used to enable the dryrun function. The dryrun function is mainly used to simulate the actual operation of the large model. After it is enabled, the memory usage, compilation information, etc. can be simulated without occupying device card. In the PyNative mode, once it is enabled, if values are fetched from the device to the host, the Python call stack log will be printed to inform users that these values are inaccurate.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore.utils import dryrun
>>> import numpy as np
>>> dryrun.set_simulation()
>>> print(os.environ.get('MS_SIMULATION_LEVEL'))
1
```

mindspore.utils.dryrun.**mock**(*mock_val*, **args*)

In the network, if some *if* branch need to use the actual execution values and the virtual execution cannot obtain them, this interface can be used to return simulated values. During actual execution, the correct results can be obtained and the execution values can be returned.

Parameters

- **mock_val** (*Union[Value, Tensor]*) – The value you want to return.
- **args** (*Union[Value, function]*) – The content you want to mock, it can be values, function and so on.

Returns

If dryrun is enabled, mock_val will be returned; otherwise, the actual execution value of args will be returned.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore.utils import dryrun
>>> import numpy as np
>>> dryrun.set_simulation()
>>> a = ms.Tensor(np.random.rand(3, 3).astype(np.float32))
>>> if dryrun.mock(True, a[0, 0] > 0.5):
...     print("return mock_val: True.")
return mock_val: True
>>>
>>> import mindspore as ms
>>> from mindspore.utils import dryrun
>>> import numpy as np
>>> a = ms.Tensor(np.ones((3, 3)).astype(np.float32))
>>> if dryrun.mock(False, a[0, 0] > 0.5):
...     print("return real execution: True.")
return real execution: True.
>>>
>>> import mindspore as ms
>>> from mindspore.utils import dryrun
>>> import numpy as np
>>> a = ms.Tensor(np.ones((3, 3)).astype(np.float32))
>>> if dryrun.mock(False, (a > 0.5).any):
...     print("return real execution: True.")
return real execution: True.
```

CHAPTER

SEVENTEEN

MINDSPORE.EXPERIMENTAL

The experimental modules.

17.1 Experimental Optimizer

API Name	Description	Supported Platforms
mindspore.experimental.optim.Optimizer	Base class for all optimizers.	Ascend GPU CPU
mindspore.experimental.optim.Adadelta	Implements Adadelta algorithm.	Ascend GPU CPU
mindspore.experimental.optim.Adagrad	Implements Adagrad algorithm.	Ascend GPU CPU
mindspore.experimental.optim.Adam	Implements Adam algorithm.	Ascend GPU CPU
mindspore.experimental.optim.Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm).	Ascend GPU CPU
mindspore.experimental.optim.AdamW	Implements Adam Weight Decay algorithm.	Ascend GPU CPU
mindspore.experimental.optim.ASGD	Implements Averaged Stochastic Gradient Descent algorithm.	Ascend GPU CPU
mindspore.experimental.optim.NAdam	Implements NAdam algorithm.	Ascend GPU CPU
mindspore.experimental.optim.RAdam	Implements RAdam algorithm.	Ascend GPU CPU
mindspore.experimental.optim.RMSprop	Implements RMSprop algorithm.	Ascend GPU CPU
mindspore.experimental.optim.Rprop	Implements Rprop algorithm.	Ascend GPU CPU
mindspore.experimental.optim.SGD	Stochastic Gradient Descent optimizer.	Ascend GPU CPU

17.1.1 mindspore.experimental.optim.Optimizer

class mindspore.experimental.optim.**Optimizer**(*params*, *defaults*)

Base class for all optimizers.

> **Warning:** This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – an iterable of *mindspore.Parameter* or dict. Specifies what Tensors should be optimized.
- **defaults** (*dict*) – a dict containing default values of optimization options (used when a parameter group doesn't specify them).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, Tensor, Parameter
>>> from mindspore import ops
>>> from mindspore.experimental import optim
>>>
>>> class MySGD(optim.Optimizer):
...     def __init__(self, params, lr):
...         defaults = dict(lr=lr)
...         super(MySGD, self).__init__(params, defaults)
...
...     def construct(self, gradients):
...          for group_id, group in enumerate(self.param_groups):
...             id = self.group_start_id[group_id]
...             for i, param in enumerate(group["params"]):
...                 next_param = param + gradients[id+i] * group["lr"]
...                 ops.assign(param, next_param)
>>>
>>> net = nn.Dense(8, 2)
>>> data = Tensor(np.random.rand(20, 8).astype(np.float32))
>>> label = Tensor(np.random.rand(20, 2).astype(np.float32))
>>>
>>> optimizer = MySGD(net.trainable_params(), 0.01)
>>> optimizer.add_param_group({"params": Parameter([0.01, 0.02])})
>>>
>>> criterion = nn.MAELoss(reduction="mean")
>>>
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = criterion(logits, label)
...     return loss, logits
>>>
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     print(loss)
>>>
>>> train_step(data, label)
```

add_param_group(*param_group*)

Add a param group to the *Optimizer.param_groups*.

Parameters

param_group (*dict*) – Specifies what Parameters should be optimized along with group specific optimization options.

17.1.2 mindspore.experimental.optim.Adadelta

class mindspore.experimental.optim.**Adadelta**(*params*, *lr=1.0*, *rho=0.9*, *eps=1e-6*, *weight_decay=0.0*, ***, *maximize=False*)

Implements Adadelta algorithm.

---

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), ρ (decay), λ (weight decay)
initialize : $v_0 \leftarrow 0$ (square avg), $u_0 \leftarrow 0$ (accumulate variables)

---

for $t = 1$ **to** ... **do**

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$if\ \lambda \neq 0$
$\quad g_t \leftarrow g_t + \lambda\theta_{t-1}$
$v_t \leftarrow v_{t-1}\rho + g_t^2(1-\rho)$
$\Delta x_t \leftarrow \frac{\sqrt{u_{t-1}+\epsilon}}{\sqrt{v_t+\epsilon}} g_t$
$u_t \leftarrow u_{t-1}\rho + \Delta x_t^2(1-\rho)$
$\theta_t \leftarrow \theta_{t-1} - \gamma\Delta x_t$

---

return θ_t

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `1.0`.
- **rho** (*float, optional*) – coefficient used for computing a running average of squared gradients. ρ in the formula above. Default: `0.9`.
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. ϵ in the formula above. Default: `1e-6`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.`.

Keyword Arguments

maximize (bool, *optional*) – maximize the params based on the objective, instead of minimizing. Default: False.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *eps* is less than or equal to 0.0.
- **ValueError** – If the *rho* is not in the range of [0, 1].
- **ValueError** – If the *weight_decay* is less than 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adadelta(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.3 mindspore.experimental.optim.Adagrad

class mindspore.experimental.optim.**Adagrad**(*params*, *lr=1e-2*, *lr_decay=0.0*, *weight_decay=0.0*, *initial_accumulator_value=0.0*, *eps=1e-10*, ***, *maximize=False*)

Implements Adagrad algorithm.

$$
\begin{aligned}
&\textbf{input} : \gamma \text{ (lr)},\ \theta_0 \text{ (params)},\ f(\theta) \text{ (objective)},\ \lambda \text{ (weight decay)},\\
&\qquad \tau \text{ (initial accumulator value)},\ \eta \text{ (lr decay)}\\
&\textbf{initialize} : state\_sum_0 \leftarrow 0\\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do}\\
&\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})\\
&\quad \tilde{\gamma} \leftarrow \gamma / (1 + (t-1)\eta)\\
&\quad \textbf{if } \lambda \neq 0\\
&\quad\quad g_t \leftarrow g_t + \lambda \theta_{t-1}\\
&\quad state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2\\
&\quad \theta_t \leftarrow \theta_{t-1} - \tilde{\gamma} \frac{g_t}{\sqrt{state\_sum_t} + \epsilon}\\
&\textbf{return } \theta_t
\end{aligned}
$$

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `1e-2`.
- **lr_decay** (*float, optional*) – learning rate decay. Default: `0.`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.`.
- **initial_accumulator_value** (*float, optional*) – the initial accumulator value. Default: `0.`.
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. Default: `1e-10`.

Keyword Arguments

maximize (*bool, optional*) – maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the learning rate decay is less than 0.
- **ValueError** – If the *weight_decay* is less than 0.
- **ValueError** – If the *initial_accumulator_value* is less than 0.0.
- **ValueError** – If the *eps* is less than 0.0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adagrad(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.4 mindspore.experimental.optim.Adam

class mindspore.experimental.optim.**Adam**(*params*, *lr=1e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=0.0*, *amsgrad=False*, ***, *maximize=False*)

Implements Adam algorithm.

The updating formulas are as follows:

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

---

return θ_t

Warning: The implementation formula of this optimizer interface is not completely consistent with that in the paper. If you want to use an interface that is completely consistent, it is recommended to use *mindspore.mint.optim.Adam*, which currently only supports Ascend. This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups
- **lr** (*Union[int, float, Tensor], optional*) –learning rate. Default: `1e-3`.
- **betas** (*Tuple[float, float], optional*) –The exponential decay rate for the moment estimations. Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) –term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `0.`.
- **amsgrad** (*bool, optional*) –whether to use the AMSGrad algorithm. Default: `False`.

Keyword Arguments

maximize (`bool, optional`) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the *lr* is not int, float or Tensor.
- **ValueError** –If the *lr* is less than 0.
- **ValueError** –If the *eps* is less than 0.0.
- **ValueError** –If the *betas* not in the range of [0, 1).
- **ValueError** –If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adam(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
```

(continues on next page)

(continued from previous page)

```
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.5 mindspore.experimental.optim.Adamax

class mindspore.experimental.optim.**Adamax**(*params*, *lr=2e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=0.0*, ***, *maximize=False*)

Implements Adamax algorithm (a variant of Adam based on infinity norm).

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
ϵ (epsilon)
initialize : $m_0 \leftarrow 0$ (first moment), $u_0 \leftarrow 0$ (infinity norm)

---

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$if\ \lambda \neq 0$
$g_t \leftarrow g_t + \lambda\theta_{t-1}$
$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
$u_t \leftarrow \max(\beta_2 u_{t-1}, |g_t| + \epsilon)$
$\theta_t \leftarrow \theta_{t-1} - \frac{\gamma m_t}{(1-\beta_1^t) u_t}$

---

return θ_t

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `2e-3`.
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square. Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.`.

Keyword Arguments

maximize (*bool, optional*) – maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.

- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *eps* is less than 0.0.
- **ValueError** – If the *weight_decay* is less than 0.
- **ValueError** – If elements of the *betas* not in the range of [0,1).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adamax(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.6 mindspore.experimental.optim.AdamW

class mindspore.experimental.optim.**AdamW**(*params*, *lr=1e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=1e-2*, *amsgrad=False*, ***, *maximize=False*)

Implements Adam Weight Decay algorithm.

---

input : γ(lr), β_1, β_2(betas), θ_0(params), $f(\theta)$(objective), ϵ (epsilon)
λ(weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $\theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right)$
  $\widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right)$
  if $amsgrad$
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right)$
  else
    $\theta_t \leftarrow \theta_t - \gamma\widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)$

---

return θ_t

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `1e-3`.
- **betas** (*Tuple[float, float], optional*) – The exponential decay rate for the moment estimations. Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.`.
- **amsgrad** (*bool, optional*) – whether to use the AMSGrad algorithm. Default: `False`.

Keyword Arguments

maximize (*bool, optional*) – maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the learning rate is not int, float or Tensor.
- **ValueError** –If the learning rate is less than 0.
- **ValueError** –If the *eps* is less than 0.0.
- **ValueError** –If the *betas* not in the range of [0, 1).
- **ValueError** –If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.AdamW(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.7 mindspore.experimental.optim.ASGD

class mindspore.experimental.optim.**ASGD**(*params*, *lr=1e-2*, *lambd=1e-4*, *alpha=0.75*, *t0=1e6*, *weight_decay=0.0*, *maximize=False*)

Implements Averaged Stochastic Gradient Descent algorithm.

> **Warning:** This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) –learning rate. Default: `1e-2`.
- **lambd** (*float, optional*) –decay term. Default: `1e-4`.
- **alpha** (*float, optional*) –power for eta update. Default: `0.75`.
- **t0** (*float, optional*) –point at which to start averaging. Default: `1e6`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `0.`.

- **maximize** (*bool, optional*) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the learning rate is not int, float or Tensor.
- **ValueError** –If the learning rate is less than 0.
- **ValueError** –If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.ASGD(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.8 mindspore.experimental.optim.NAdam

class mindspore.experimental.optim.**NAdam**(*params*, *lr=2e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=0.0*, *momentum_decay=4e-3*)

Implements NAdam algorithm.

> **Warning:** This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) –learning rate. Default: `2e-3`.

- **betas** (*Tuple[float, float], optional*) –coefficients used for computing running averages of gradient and its square. Default: `(0.9, 0.999)`.
- **eps** (*float, optional*) –term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `0.`.
- **momentum_decay** (*float, optional*) –momentum momentum_decay. Default: `4e-3`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the learning rate is not int, float or Tensor.
- **ValueError** –If the learning rate is less than 0.
- **ValueError** –If the *eps* is less than 0.0.
- **ValueError** –If the *weight_decay* is less than 0.
- **ValueError** –If the *momentum_decay* is less than 0.
- **ValueError** –If elements of *betas* not in the range of [0, 1).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.NAdam(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.9 mindspore.experimental.optim.RAdam

class mindspore.experimental.optim.**RAdam**(*params*, *lr=1e-3*, *betas=(0.9, 0.999)*, *eps=1e-8*, *weight_decay=0.0*)

Implements RAdam algorithm.

Input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective), λ (weightdecay), ϵ (epsilon)

Initialize : $\begin{cases} m_0 \leftarrow 0 \text{ (first moment)} \\ v_0 \leftarrow 0 \text{ (second moment)} \\ \rho_\infty \xleftarrow{\text{def}} \dfrac{2}{1-\beta_2} - 1 \end{cases}$

For $t = 1$ to $\ldots$ do :

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

If $\lambda \neq 0$:

$g_t \leftarrow g_t + \lambda\theta_{t-1}$

$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$

$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$

$\widehat{m_t} \leftarrow \dfrac{m_t}{1-\beta_1^t}$

Let $\rho_t' = 2t\beta_2^t/(1-\beta_2^t)$ (auxiliary variable)

$\rho_t \leftarrow \rho_\infty - \rho_t'$

If $\rho_t > 5$:

$l_t \leftarrow \dfrac{\sqrt{1-\beta_2^t}}{\sqrt{v_t}+\epsilon}$

$r_t \leftarrow \sqrt{\dfrac{(\rho_t-4)(\rho_t-2)\rho_\infty}{(\rho_\infty-4)(\rho_\infty-2)\rho_t}}$

$\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} r_t l_t$

Else :

$\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t}$

Return : θ_t

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: 1e-3.
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square. Default: (0.9, 0.999).

- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.0`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *eps* is less than 0.0.
- **ValueError** – If the *weight_decay* is less than 0.
- **ValueError** – If elements of *betas* not in the range of [0, 1).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.RAdam(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.10 mindspore.experimental.optim.RMSprop

class mindspore.experimental.optim.**RMSprop**(*params*, *lr=1e-2*, *alpha=0.99*, *eps=1e-8*, *weight_decay=0.0*, *momentum=0.0*, *centered=False*, *maximize=False*)

Implements RMSprop algorithm.

> **Warning:** This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `1e-2`.
- **alpha** (*float, optional*) – smoothing constant. Default: `0.99`.
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability. Default: `1e-8`.
- **weight_decay** (*float, optional*) – weight decay (L2 penalty). Default: `0.`.
- **momentum** (*float, optional*) – momentum factor. Default: `0.`.
- **centered** (*bool, optional*) – if `True`, compute the centered RMSProp, the gradient is normalized by an estimation of its variance. Default: `False`.
- **maximize** (*bool, optional*) – maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *momentum* is less than 0.0.
- **ValueError** – If the *alpha* is less than 0.0.
- **ValueError** – If the *eps* is less than 0.0.
- **ValueError** – If the *weight_decay* is less than 0.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.RMSprop(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.11 mindspore.experimental.optim.Rprop

class mindspore.experimental.optim.**Rprop**(*params*, *lr=1e-2*, *etas=(0.5, 1.2)*, *step_sizes=(1e-6, 50)*, ***, *maximize=False*)

Implements Rprop algorithm.

> **Warning:** This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) – list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor], optional*) – learning rate. Default: `1e-2`.
- **etas** (*Tuple[float, float], optional*) – pair of (etaminus, etaplus), that are multiplicative increase and decrease factors. Default: `(0.5, 1.2)`
- **step_sizes** (*Tuple[float, float], optional*) – a pair of minimal and maximal allowed step sizes. Default: `(1e-6, 50)`

Keyword Arguments

maximize (`bool, optional`) – maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** – If the learning rate is not int, float or Tensor.
- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *etas[1]* is less than or equal to 1.0.
- **ValueError** – If the *etas[0]* not in the range of 0-1.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Rprop(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
```

(continues on next page)

(continued from previous page)

```
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.12 mindspore.experimental.optim.SGD

class mindspore.experimental.optim.**SGD**(*params*, *lr*, *momentum=0*, *dampening=0*, *weight_decay=0.0*, *nesterov=False*, ***, *maximize=False*)

Stochastic Gradient Descent optimizer.

$$v_{t+1} = u * v_t + gradient * (1 - dampening)$$

If nesterov is True:

$$p_{t+1} = p_t - lr * (gradient + u * v_{t+1})$$

If nesterov is False:

$$p_{t+1} = p_t - lr * v_{t+1}$$

To be noticed, for the first step, $v_{t+1} = gradient$.

Here: where p, v and u denote the parameters, accum, and momentum respectively.

Warning: This is an experimental optimizer API that is subject to change. This module must be used with lr scheduler module in LRScheduler Class .

Parameters

- **params** (*Union[list(Parameter), list(dict)]*) –list of parameters to optimize or dicts defining parameter groups.
- **lr** (*Union[int, float, Tensor]*) –learning rate.
- **momentum** (*Union[int, float], optional*) –momentum factor. Default: `0`.
- **weight_decay** (*float, optional*) –weight decay (L2 penalty). Default: `0`..
- **dampening** (*Union[int, float], optional*) –dampening for momentum. Default: `0`.
- **nesterov** (*bool, optional*) –enables Nesterov momentum. Default: `False`.

Keyword Arguments

maximize (*bool, optional*) –maximize the params based on the objective, instead of minimizing. Default: `False`.

Inputs:

- **gradients** (tuple[Tensor]) - The gradients of *params*.

Raises

- **ValueError** –If the learning rate is not int, float or Tensor.

- **ValueError** – If the learning rate is less than 0.
- **ValueError** – If the *momentum* or *weight_decay* value is less than 0.0.
- **ValueError** – If the *momentum*, *dampening* or *weight_decay* value is not int or float.
- **ValueError** – If the *nesterov* and *maximize* is not bool.
- **ValueError** – If the *nesterov* is true, *momentum* is not positive or *dampening* is not 0.0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.SGD(net.trainable_params(), lr=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
```

17.1.13 LRScheduler Class

The dynamic learning rates in this module are all subclasses of LRScheduler, this module should be used with optimizers in mindspore.experimental.optim, pass the optimizer instance to a LRScheduler when used. During the training process, the LRScheduler subclass dynamically changes the learning rate by calling the *step* method.

```
import mindspore
from mindspore import nn
from mindspore.experimental import optim

# Define the network structure of LeNet5. Refer to
# https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py

net = LeNet5()
loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
optimizer = optim.Adam(net.trainable_params(), lr=0.05)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.1)
def forward_fn(data, label):
    logits = net(data)
    loss = loss_fn(logits, label)
    return loss, logits
grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
```

(continues on next page)

(continued from previous page)

```
def train_step(data, label):
    (loss, _), grads = grad_fn(data, label)
    optimizer(grads)
    return loss
for epoch in range(6):
    # Create the dataset taking MNIST as an example. Refer to
    # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py

    for data, label in create_dataset(need_download=False):
        train_step(data, label)
    scheduler.step()
```

API Name	Description	Supported Platforms
mindspore.experimental.optim.lr_scheduler.LRScheduler	Basic class of learning rate schedule.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.ConstantLR	Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total_iters.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.CosineAnnealingLR	Set the learning rate of each parameter group using a cosine annealing lr schedule.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.CosineAnnealingWarmRestarts	Set the learning rate of each parameter group using a cosine annealing warm restarts schedule.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.CyclicLR	Sets the learning rate of each parameter group according to cyclical learning rate policy (CLR).	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.ExponentialLR	For each epoch, the learning rate decays exponentially, multiplied by gamma.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.LambdaLR	Sets the learning rate of each parameter group to the initial lr times a given function.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.LinearLR	Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: total_iters.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.MultiplicativeLR	Multiply the learning rate of each parameter group by the factor given in the specified function.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.MultiStepLR	Multiply the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.PolynomialLR	For each epoch, the learning rate is adjusted by polynomial fitting.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.ReduceLROnPlateau	Reduce learning rate when a metric has stopped improving.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.SequentialLR	Concatenate multiple learning rate adjustment strategies in *schedulers* in sequence, switching to the next learning rate adjustment strategy at *milestone*.	Ascend GPU CPU
mindspore.experimental.optim.lr_scheduler.StepLR	Decays the learning rate of each parameter group by gamma every step_size epochs.	Ascend GPU CPU

mindspore.experimental.optim.lr_scheduler.LRScheduler

class mindspore.experimental.optim.lr_scheduler.**LRScheduler**(*optimizer*, *last_epoch=- 1*)

Basic class of learning rate schedule.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –The optimizer instance.
- **last_epoch** (*int, optional*) –The number of times the *step()* method of the current learning rate adjustment strategy has been executed. Default: `-1`.

Raises

- **TypeError** –If *optimizer* does not satisfy the type requirement.
- **KeyError** –If *last_epoch* != -1 and `'initial_lr'` not in param groups.
- **ValueError** –if *last_epoch* is not int.
- **ValueError** –If *last_epoch* is not greater than -1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>>
>>> class ConstantLR(optim.lr_scheduler.LRScheduler):
...     def __init__(self, optimizer, factor=0.5, total_iters=3, last_epoch=-1):
...         self.factor = factor
...         self.total_iters = total_iters
...         super(ConstantLR, self).__init__(optimizer, last_epoch)
...
...     def get_lr(self):
...         if self.last_epoch == 0:
...             return [lr * self.factor for lr in self._last_lr]
...         if self.last_epoch != self.total_iters:
...             return [lr * 1. for lr in self._last_lr]
...         return [lr / self.factor for lr in self._last_lr]
>>>
>>> net = nn.Dense(8, 2)
>>> optimizer = optim.SGD(net.trainable_params(), 0.01)
>>> scheduler = ConstantLR(optimizer)
>>> for i in range(4):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.005)]
[Tensor(shape=[], dtype=Float32, value= 0.005)]
```

(continues on next page)

(continued from previous page)

```
[Tensor(shape=[], dtype=Float32, value= 0.01)]
[Tensor(shape=[], dtype=Float32, value= 0.01)]
```

get_last_lr()
Return last computed learning rate by current scheduler.

step(*epoch=None*)
Get the current learning rate and change the learning rate.

Parameters

epoch (*int, optional*) –The index of the last epoch. Default: None.

mindspore.experimental.optim.lr_scheduler.ConstantLR

class mindspore.experimental.optim.lr_scheduler.**ConstantLR**(*optimizer*, *factor=1.0 / 3*, *total_iters=5*, *last_epoch=- 1*)

Decays the learning rate of each parameter group by a small constant factor until the number of epoch reaches a pre-defined milestone: total_iters. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **factor** (*float, optional*) –The factor number multiplied learning rate. Default: 1./3.
- **total_iters** (*int, optional*) –The number of steps that the scheduler decays the learning rate, when the *last_epoch* reach *total_iters*, restore the learning rate. Default: 5.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: -1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> net = nn.Dense(2, 3)
>>> optimizer = optim.Adam(net.trainable_params(), 0.05)
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.025   if epoch <4
>>> # lr = 0.05    if epoch >= 4
>>> scheduler = optim.lr_scheduler.ConstantLR(optimizer, factor=0.5, total_iters=4)
>>> for i in range(6):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.025)]
```

(continues on next page)

(continued from previous page)

```
[Tensor(shape=[], dtype=Float32, value= 0.025)]
[Tensor(shape=[], dtype=Float32, value= 0.025)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
```

mindspore.experimental.optim.lr_scheduler.CosineAnnealingLR

class mindspore.experimental.optim.lr_scheduler.**CosineAnnealingLR**(*optimizer*, *T_max*, *eta_min=0.0*, *last_epoch=- 1*)

Set the learning rate of each parameter group using a cosine annealing lr schedule. Where η_{max} is set to the initial lr, η_{min} is the minimum value for learning rate, η_t is the current learning rate, T_{max} is iteration number of cosine function, and T_{cur} is the number of epochs since the last restart in SGDR.

$$
\begin{aligned}
&\eta_t && = \eta_{min} + \tfrac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\tfrac{T_{cur}}{T_{max}}\pi\right)\right), \\
&T_{cur} \neq (2k+1)T_{max}; && \\
&\eta_{t+1} && = \eta_t + \tfrac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\tfrac{1}{T_{max}}\pi\right)\right), \\
&T_{cur} = (2k+1)T_{max}. &&
\end{aligned}
$$

For more details, please refer to: SGDR: Stochastic Gradient Descent with Warm Restarts .

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **T_max** (*int*) –Maximum number of iterations.
- **eta_min** (*float, optional*) –Minimum learning rate. Default: `0.0`.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.experimental import optim
>>> from mindspore import nn
>>> net = nn.Dense(3, 2)
>>> optimizer = optim.SGD(net.trainable_params(), lr=0.1, momentum=0.9)
>>> scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=2)
>>>
>>> for i in range(6):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.05)]
```

(continues on next page)

(continued from previous page)

```
[Tensor(shape=[], dtype=Float32, value= 0)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0.1)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0)]
```

mindspore.experimental.optim.lr_scheduler.CosineAnnealingWarmRestarts

class mindspore.experimental.optim.lr_scheduler.**CosineAnnealingWarmRestarts**(*optimizer*, *T_0*, *T_mult=1*, *eta_min=0*, *last_epoch=- 1*)

Set the learning rate of each parameter group using a cosine annealing warm restarts schedule.

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

Where η_{max} is set to the initial lr, η_{min} is the minimum value for learning rate, η_t is the current learning rate, T_0 is the number of iterations for the first restar, T_i is the current number of iterations between two warm restarts in SGDR, T_{cur} is the number of epochs since the last restart in SGDR.

When $T_{cur} = T_i$, set $\eta_t = \eta_{min}$. When $T_{cur} = 0$ after restart, set $\eta_t = \eta_{max}$.

For more details, please refer to: SGDR: Stochastic Gradient Descent with Warm Restarts.

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **T_0** (*int*) –Number of iterations for the first restart.
- **T_mult** (*int, optional*) –A factor increases T_i after a restart. Default: `1`.
- **eta_min** (*Union(float, int), optional*) –Minimum learning rate. Default: `0`.
- **last_epoch** (*int, optional*) –The number of times the *step()* method of the current learning rate adjustment strategy has been executed. Default: `-1`.

Raises

- **ValueError** –*T_0* is less than or equal than 0 or not an int.
- **ValueError** –*T_mult* is less than or equal than 1 or not an int.
- **ValueError** –*eta_min* is not int or float.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.experimental import optim
>>> from mindspore import nn
>>> net = nn.Dense(3, 2)
>>> optimizer = optim.SGD(net.trainable_params(), lr=0.1, momentum=0.9)
>>> scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, 2)
>>> iters = 3
>>> for epoch in range(2):
...     for i in range(iters):
...         scheduler.step(epoch + i / iters)
...         current_lr = scheduler.get_last_lr()
...         print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.1)]
[Tensor(shape=[], dtype=Float32, value= 0.0933013)]
[Tensor(shape=[], dtype=Float32, value= 0.075)]
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0.025)]
[Tensor(shape=[], dtype=Float32, value= 0.00669873)]
```

step(*epoch=None*)

Get the current learning rate and change the learning rate.

Parameters

epoch (*int, optional*) – The index of the last epoch. Default: `None`.

mindspore.experimental.optim.lr_scheduler.CyclicLR

class mindspore.experimental.optim.lr_scheduler.**CyclicLR**(*optimizer*, *base_lr*, *max_lr*, *step_size_up=2000*, *step_size_down=None*, *mode='triangular'*, *gamma=1.*, *scale_fn=None*, *scale_mode='cycle'*, *last_epoch=- 1*)

Sets the learning rate of each parameter group according to cyclical learning rate policy (CLR). The policy cycles the learning rate between two boundaries with a constant frequency, as detailed in the paper Cyclical Learning Rates for Training Neural Networks. The distance between the two boundaries can be scaled on a per-iteration or per-cycle basis.

This class has three built-in policies, as put forth in the paper:

- "triangular": A basic triangular cycle without amplitude scaling.
- "triangular2": A basic triangular cycle that scales initial amplitude by half each cycle.
- "exp_range": A cycle that scales initial amplitude by $\text{gamma}^{\text{cycle iterations}}$ at each cycle iteration.

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) – Wrapped optimizer.
- **base_lr** (*Union(float, list)*) – Initial learning rate which is the lower boundary in the cycle for each parameter group.

- **max_lr** (*Union(float, list)*) –Upper learning rate boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_lr - base_lr). The lr at any cycle is the sum of base_lr and some scaling of the amplitude.
- **step_size_up** (*int, optional*) –Number of training iterations in the increasing half of a cycle. Default: `2000`.
- **step_size_down** (*int, optional*) –Number of training iterations in the decreasing half of a cycle. If step_size_down is None, it is set to step_size_up. Default: `None`.
- **mode** (*str, optional*) –One of {triangular, triangular2, exp_range}. Values correspond to policies detailed above. If scale_fn is not None, this argument is ignored. Default: `'triangular'`.
- **gamma** (*float, optional*) –Constant in 'exp_range' scaling function: gamma**(cycle iterations). Default: `1.0`.
- **scale_fn** (*function, optional*) –Custom scaling policy defined by a single argument lambda function, where 0 <= scale_fn(x) <= 1 for all x >= 0. If specified, then 'mode' is ignored. Default: `None`.
- **scale_mode** (*str, optional*) –{'cycle', 'iterations'}. Defines whether scale_fn is evaluated on cycle number or cycle iterations (training iterations since start of cycle). Illegal inputs will use `'iterations'` by defaults. Default: `'cycle'`.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Raises

- **ValueError** –When *base_lr* is list or tuple, the length of it is not equal to the number of param groups.
- **ValueError** –When *max_lr* is list or tuple, the length of it is not equal to the number of param groups.
- **ValueError** –*mode* is not in [`'triangular'`, `'triangular2'`, `'exp_range'`] and *scale_fn* is None.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.experimental import optim
>>> from mindspore import nn
>>> net = nn.Dense(3, 2)
>>> optimizer = optim.SGD(net.trainable_params(), lr=0.1, momentum=0.9)
>>> scheduler = optim.lr_scheduler.CyclicLR(optimizer, base_lr=0.01, max_lr=0.1)
>>> expect_list = [[0.010045], [0.01009], [0.010135], [0.01018], [0.010225]]
>>>
>>> for i in range(5):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.010045)]
[Tensor(shape=[], dtype=Float32, value= 0.01009)]
[Tensor(shape=[], dtype=Float32, value= 0.010135)]
[Tensor(shape=[], dtype=Float32, value= 0.01018)]
[Tensor(shape=[], dtype=Float32, value= 0.010225)]
```

mindspore.experimental.optim.lr_scheduler.ExponentialLR

class mindspore.experimental.optim.lr_scheduler.**ExponentialLR**(*optimizer*, *gamma*, *last_epoch=- 1*)

For each epoch, the learning rate decays exponentially, multiplied by gamma. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **gamma** (*float*) –Learning rate scaling factor.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: -1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(16 * 5 * 5, 120)
...     def construct(self, x):
...         return self.fc(x)
>>> net = Net()
>>> optimizer = optim.Adam(net.trainable_params(), 0.01)
>>> scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.5)
>>> for i in range(3):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.005)]
[Tensor(shape=[], dtype=Float32, value= 0.0025)]
[Tensor(shape=[], dtype=Float32, value= 0.00125)]
```

mindspore.experimental.optim.lr_scheduler.LambdaLR

class mindspore.experimental.optim.lr_scheduler.**LambdaLR**(*optimizer*, *lr_lambda*, *last_epoch=- 1*)

Sets the learning rate of each parameter group to the initial lr times a given function. When last_epoch=-1, sets initial lr as lr.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.

- **lr_lambda** (*Union(function, list)*) –A function which computes a multiplicative factor given a parameter *last_epoch*, or a list of such functions, one for each group in *optimizer.param_groups*.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: -1.

Raises

ValueError –If the length of *lr_lambda* is not equal to the number of param groups.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> net = nn.Dense(2, 3)
>>> optimizer = optim.Adam(net.trainable_params(), 0.01)
>>> lmbda = lambda epoch: 0.9 ** epoch
>>> scheduler = optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=[lmbda])
>>> for i in range(3):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.009)]
[Tensor(shape=[], dtype=Float32, value= 0.0081)]
[Tensor(shape=[], dtype=Float32, value= 0.00729)]
```

mindspore.experimental.optim.lr_scheduler.LinearLR

class mindspore.experimental.optim.lr_scheduler.**LinearLR**(*optimizer*, *start_factor=1.0 / 3*, *end_factor=1.0*, *total_iters=5*, *last_epoch=- 1*)

Decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a pre-defined milestone: total_iters. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **start_factor** (*float, optional*) –The number we multiply learning rate in the first epoch. The multiplication factor changes towards *end_factor* in the following epochs. Default: 1.0 /3.
- **end_factor** (*float, optional*) –The number we multiply learning rate at the end of linear changing process. Default: 1.0.
- **total_iters** (*int, optional*) –The number of iterations that multiplicative factor reaches to 1. Default: 5.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: -1.

Raises

- **ValueError** –If *start_factor* is not in the range of (0, 1].

- **ValueError** – If *end_factor* is not in the range of [0, 1].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adam(net.trainable_params(), lr=0.05)
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.025    if epoch == 0
>>> # lr = 0.03125  if epoch == 1
>>> # lr = 0.0375   if epoch == 2
>>> # lr = 0.04375  if epoch == 3
>>> # lr = 0.05    if epoch >= 4
>>> scheduler = optim.lr_scheduler.LinearLR(optimizer, start_factor=0.5, total_iters=4)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
>>> for epoch in range(5):
...     # Create the dataset taking MNIST as an example. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
...     for data, label in create_dataset():
...         train_step(data, label)
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
```

mindspore.experimental.optim.lr_scheduler.MultiplicativeLR

class mindspore.experimental.optim.lr_scheduler.**MultiplicativeLR**(*optimizer*, *lr_lambda*, *last_epoch=-1*)

Multiply the learning rate of each parameter group by the factor given in the specified function. When last_epoch=-1, sets initial lr as lr.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) – Wrapped optimizer.

- **lr_lambda** (*Union(function, list)*) –A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> net = nn.Dense(2, 3)
>>> optimizer = optim.Adam(net.trainable_params(), 0.01)
>>> lmbda = lambda epoch: 0.95
>>> scheduler = optim.lr_scheduler.MultiplicativeLR(optimizer, lr_lambda=lmbda)
>>> for i in range(3):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.0095)]
[Tensor(shape=[], dtype=Float32, value= 0.009025)]
[Tensor(shape=[], dtype=Float32, value= 0.00857375)]
```

mindspore.experimental.optim.lr_scheduler.MultiStepLR

class mindspore.experimental.optim.lr_scheduler.**MultiStepLR**(*optimizer*, *milestones*, *gamma=0.1*, *last_epoch=- 1*)

Multiply the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestones. Notice that such change can happen simultaneously with other changes to the learning rate from outside this scheduler. When last_epoch=-1, sets initial lr as lr.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **milestones** (*list*) –List of epoch indices. When *last_epoch* reach the milestone, multiply the learning rate of each parameter group by *gamma*.
- **gamma** (*float, optional*) –Multiplicative factor of learning rate decay. Default: `0.1`.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Raises

- **TypeError** –If the *milestones* is not list.
- **TypeError** –If elements of the *milestones* are not int.
- **TypeError** –If the *gamma* is not float.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> net = nn.Dense(2, 3)
>>> optimizer = optim.Adam(net.trainable_params(), 0.05)
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 2
>>> # lr = 0.005    if 2 <= epoch < 4
>>> # lr = 0.0005   if epoch >= 4
>>> scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[2,4], gamma=0.1)
>>> for i in range(6):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.05)]
[Tensor(shape=[], dtype=Float32, value= 0.005)]
[Tensor(shape=[], dtype=Float32, value= 0.005)]
[Tensor(shape=[], dtype=Float32, value= 0.0005)]
[Tensor(shape=[], dtype=Float32, value= 0.0005)]
[Tensor(shape=[], dtype=Float32, value= 0.0005)]
```

mindspore.experimental.optim.lr_scheduler.PolynomialLR

class mindspore.experimental.optim.lr_scheduler.**PolynomialLR**(*optimizer*, *total_iters=5*, *power=1.0*, *last_epoch=- 1*)

For each epoch, the learning rate is adjusted by polynomial fitting. When the epoch is greater than or equal to *total_iters* , the learning rate is `0` . Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

The polynomial formula for learning rate calculation is as follows:

$$factor = (\frac{1.0 - \frac{last\_epoch}{total\_iters}}{1.0 - \frac{last\_epoch - 1.0}{total\_iters}})^{power}$$
$$lr = lr \times factor$$

Warning: This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **total_iters** (*int, optional*) –The number of iterations adjusting learning rate by polynomial fitting. Default: `5`.
- **power** (*float, optional*) –Power of polynomial. Default: `1.0`.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.fc = nn.Dense(16 * 5 * 5, 120)
...     def construct(self, x):
...         return self.fc(x)
>>> net = Net()
>>> optimizer = optim.Adam(net.trainable_params(), 0.01)
>>> scheduler = optim.lr_scheduler.PolynomialLR(optimizer)
>>> for i in range(6):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.008)]
[Tensor(shape=[], dtype=Float32, value= 0.006)]
[Tensor(shape=[], dtype=Float32, value= 0.004)]
[Tensor(shape=[], dtype=Float32, value= 0.002)]
[Tensor(shape=[], dtype=Float32, value= 0)]
[Tensor(shape=[], dtype=Float32, value= 0)]
```

mindspore.experimental.optim.lr_scheduler.ReduceLROnPlateau

class mindspore.experimental.optim.lr_scheduler.**ReduceLROnPlateau**(*optimizer*, *mode='min'*, *factor=0.1*, *patience=10*, *threshold=1e-4*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-8*)

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. The scheduler reads the metrics *metrics* during execution and adjusts the learning rate via the *step* method if the metrics do not improve within *patience* cycles.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **mode** (*str, optional*) –Trigger mode that triggers a reduction in learning rate when the monitoring metrics are at their *min* / *max* point. In *min* mode, lr will be reduced when the quantity monitored has stopped decreasing; in *max* mode it will be reduced when the quantity monitored has stopped increasing. Default: `'min'`.
- **factor** (*float, optional*) –Factor by which the learning rate will be reduced. Default: `0.1`.
- **patience** (*int, optional*) –Number of epochs with no improvement after which learning rate will be reduced. For example, if *patience* = 2, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn't improved then. Default: `10`.

- **threshold** (*float, optional*) – Threshold for measuring the new optimum, to only focus on significant changes. Default: `1e-4`.
- **threshold_mode** (*str, optional*) – A mode for measuring indicators of change for the better. One of *rel*, *abs*. Default: `'rel'`.

  Assume that *best* represents the best value of the current performance metric.

  - In `'rel'` mode, the indicator is compared to a *threshold* in proportional form:
    - When *mode* is `'max'`, the indicator is considered better if it exceeds best * (1 + threshold).
    - When *mode* is `'min'`, the indicator is considered better if it is lower than best * (1 - threshold).
  - In `'abs'` mode, the indicator is compared to *threshold* in absolute value form:
    - When *mode* is `'max'`, the indicator is considered better if it exceeds best + threshold.
    - When *mode* is `'min'`, the indicator is considered better if it is lower than best - threshold.
- **cooldown** (*int, optional*) – Number of epochs to wait before resuming normal operation after lr has been reduced. Default: `0`.
- **min_lr** (*Union(float, list), optional*) – A scalar or a list of scalars. A lower bound on the learning rate of all param groups or each group respectively. Default: `0`.
- **eps** (*float, optional*) – Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: `1e-8`.

Raises

- **ValueError** – *factor* is greater than or equal to 1.0.
- **TypeError** – *optimizer* is not an *Optimizer*.
- **ValueError** – When *min_lr* is a list or tuple, the length of it is not equal to the number of param groups.
- **ValueError** – *mode* is neither `'min'` nor `'max'`.
- **ValueError** – *threshold_mode* is neither `'rel'` nor `'abs'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.experimental import optim
>>> from mindspore import nn
>>> net = nn.Dense(3, 2)
>>> optimizer = optim.Adam(net.trainable_params(), 0.1)
>>> scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=0)
>>> metrics = [1, 1.5, 1.8, 0.4, 0.5]
>>> for i in range(5):
...     scheduler.step(metrics[i])
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.1)]
[Tensor(shape=[], dtype=Float32, value= 0.01)]
[Tensor(shape=[], dtype=Float32, value= 0.001)]
[Tensor(shape=[], dtype=Float32, value= 0.001)]
[Tensor(shape=[], dtype=Float32, value= 0.0001)]
```

get_last_lr()
Return last computed learning rate by current scheduler.

in_cooldown()
Whether in cooldown period.

step(*metrics*)
Get the current learning rate and change the learning rate.

Parameters

metrics (*Union(int, float)*) –the evaluation metrics.

mindspore.experimental.optim.lr_scheduler.SequentialLR

class mindspore.experimental.optim.lr_scheduler.**SequentialLR**(*optimizer*, *schedulers*, *milestones*, *last_epoch=- 1*)

Concatenate multiple learning rate adjustment strategies in *schedulers* in sequence, switching to the next learning rate adjustment strategy at *milestone*.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **schedulers** (list[*mindspore.experimental.optim.lr_scheduler.LRScheduler*]) –List of learning rate schedulers.
- **milestones** (*list*) –List of integers of milestone points, sets which learning rate adjustment strategy is invoked for each epoch.
- **last_epoch** (*int, optional*) –The number of times the *step()* method of the current learning rate adjustment strategy has been executed. Default: `-1`.

Raises

- **ValueError** –The optimizer in *schedulers* is different from the *optimizer* passed in.
- **ValueError** –The optimizer in *schedulers* is different from the optimizer of *schedulers[0]*.
- **ValueError** –Length of *milestones* is not equal to length of *schedulers* minus 1.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.experimental import optim
>>> from mindspore import nn
>>> net = nn.Dense(3, 2)
>>> optimizer = optim.Adam(net.trainable_params(), 0.1)
>>> scheduler1 = optim.lr_scheduler.ConstantLR(optimizer, factor=0.1, total_iters=2)
>>> scheduler2 = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)
>>> scheduler = optim.lr_scheduler.SequentialLR(optimizer, schedulers=[scheduler1,␣
↪scheduler2], milestones=[2])
>>> for i in range(6):
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
...     print(current_lr)
[Tensor(shape=[], dtype=Float32, value= 0.01)]
[Tensor(shape=[], dtype=Float32, value= 0.1)]
[Tensor(shape=[], dtype=Float32, value= 0.09)]
[Tensor(shape=[], dtype=Float32, value= 0.081)]
[Tensor(shape=[], dtype=Float32, value= 0.0729)]
[Tensor(shape=[], dtype=Float32, value= 0.06561)]
```

get_last_lr()
Return last computed learning rate by current scheduler.

step()
Get the current learning rate and change the learning rate.

mindspore.experimental.optim.lr_scheduler.StepLR

class mindspore.experimental.optim.lr_scheduler.**StepLR**(*optimizer*, *step_size*, *gamma=0.1*, *last_epoch=- 1*)
Decays the learning rate of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler.

> **Warning:** This is an experimental lr scheduler module that is subject to change. This module must be used with optimizers in Experimental Optimizer .

Parameters

- **optimizer** (*mindspore.experimental.optim.Optimizer*) –Wrapped optimizer.
- **step_size** (*int*) –Period of learning rate decay.
- **gamma** (*float, optional*) –Multiplicative factor of learning rate decay. Default: `0.5`.
- **last_epoch** (*int, optional*) –The index of the last epoch. Default: `-1`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import nn
>>> from mindspore.experimental import optim
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True)
>>> optimizer = optim.Adam(net.trainable_params(), lr=0.05)
>>> # Assuming optimizer uses lr = 0.05 for all groups
>>> # lr = 0.05     if epoch < 2
>>> # lr = 0.005    if 2 <= epoch < 4
>>> # lr = 0.0005   if 4 <= epoch < 6
>>> scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.1)
>>> def forward_fn(data, label):
...     logits = net(data)
...     loss = loss_fn(logits, label)
...     return loss, logits
>>> grad_fn = mindspore.value_and_grad(forward_fn, None, optimizer.parameters, has_aux=True)
>>> def train_step(data, label):
...     (loss, _), grads = grad_fn(data, label)
...     optimizer(grads)
...     return loss
>>> for epoch in range(6):
...     # Create the dataset taking MNIST as an example. Refer to
...     # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
...     for data, label in create_dataset():
...         train_step(data, label)
...     scheduler.step()
...     current_lr = scheduler.get_last_lr()
```

17.2 Experimental EmbeddingService

mindspore.experimental.es.EmbeddingService	ES(EmbeddingService) feature can support model training and inference for PS embedding and data_parallel embedding, and provide unified embedding management, storage, and computing capabilities for training and inference.
mindspore.experimental.es.EsEmbeddingLookup	Look up a PS embedding.
mindspore.experimental.es.ESEmbeddingSmallTableLookup	Look up a data_parallel embedding.

17.2.1 mindspore.experimental.es.EmbeddingService

class mindspore.experimental.es.**EmbeddingService**

ES(EmbeddingService) feature can support model training and inference for PS embedding and data_parallel embedding, and provide unified embedding management, storage, and computing capabilities for training and inference. PS embedding refer to tables that vocab_size more than 100,000, and recommended to store them on the Parameter Server (PS). Data_parallel embedding refer to tables that vocab_size less than 100,000, and recommended to store them on device.

Currently, ES feature can only create one instance of EmbeddingService object.

Warning: This is an experimental EmbeddingService API that is subject to change.

Note: This API needs to call *mindspore.communication.init()* before, and it can take effect after the dynamic networking is completed.

Raises

- **ValueError** –If the ESCLUSTER_CONFIG_PATH environment variable is not set during object instantiation.
- **ValueError** –If the number of each server ParameterServer configured in ESCLUSTER_CONFIG_PATH configuration file exceeds four.
- **ValueError** –If the number of ParameterServer configured in ESCLUSTER_CONFIG_PATH configuration file exceeds four.

Supported Platforms:

`Atlas A2 training series products`

completion_key(*completion_key*, *mask=True*)

Init completion key option for each PS embedding.

Parameters

- **completion_key** (*int*) –The value for completion key.
- **mask** (*bool, optional*) –Whether to update completion key. If set to false, it will not be updated, and default is `True`.

Returns

CompletionKeyOption object.

Raises

- **TypeError** –If the type of "completion_key" is not int.
- **TypeError** –If the type of "mask" is not bool.

counter_filter(*filter_freq*, *default_key=None*, *default_value=None*)

Init counter filter option for each PS embedding.

Note: This feature only supports training mode. When user set counter filter option in train mode and then eval, the default value can be used for the key that cannot be look up in eval.

Parameters

- **filter_freq** (*int*) –The frequency threshold value for feature admission.
- **default_key** (*int, optional*) –The key that number of occurrences does not reach the threshold, return value of *default_key* as the corresponding value when look up embedding, and default is `None`.
- **default_value** (*Union[int, float], optional*) –The key that number of occurrences does not reach the threshold, return default value which length value is embedding dim, and default is `None`.

Returns

CounterFilter object.

Raises

- **TypeError** –If the type of "filter_freq" is not int.
- **ValueError** –If the value of "filter_freq" is less than 0.
- **ValueError** –If the values of "default_key" and "default_value" are None.
- **ValueError** –If neither of the values of "default_key" and "default_value" are None.
- **TypeError** –If the value of "default_key" is None and the type of "default_value" is neither int nor float.
- **TypeError** –If the value of "default_value" is None and the type of "default_key" is not int.

embedding_ckpt_export(*file_path*, *trainable_var*)

Export the embedding table and optimizer parameters of each PS embedding, and export embedding of a data_parallel embedding.

Note: This function can only be executed by rank 0. Need to call *mindspore.experimental.es.EmbeddingService.embedding_variable_option()* to set evict_option for each PS embedding before export.

Parameters

- **file_path** (*str*) –The path to export embedding ckpt, and the last character cannot be `"/"`.
- **trainable_var** (*list[parameter]*) –The list of data_parallel embedding parameter.

Returns

The output of EmbeddingComputeVarExport operator and data_parallel embedding export result.

embedding_ckpt_import(*file_path*)

Import embedding and ckpt file from file path.

Parameters

file_path (*str*) –The path to import embedding and ckpt, and the last character cannot be `"/"`.

Returns

The output of EmbeddingComputeVarImport operator and data_parallel embedding import result.

embedding_evict(*steps_to_live*)

Embedding evict for all PS embedding.

Parameters

steps_to_live (*int*) –The steps set for evict key.

Returns

The output of ESEmbeddingTableEvict op.

Raises

- **TypeError** –If the type of "steps_to_live" is not int.
- **ValueError** –If the value of "steps_to_live" is not greater than 0.

embedding_init(*name*, *init_vocabulary_size*, *embedding_dim*, *max_feature_count=None*, *initializer=Uniform(scale=0.01)*, *embedding_type='PS'*, *ev_option=None*, *multihot_lens=None*, *optimizer=None*, *allow_merge=False*, *optimizer_param=None*, *mode='train'*)

Init for PS embedding and data_parallel embedding.

Parameters

- **name** (*str*) –The embedding table name.
- **init_vocabulary_size** (*int*) –The size of embedding table.
- **embedding_dim** (*int*) –The embedding dim of data in embedding table.
- **max_feature_count** (*int, optional*) –The count of keys when look up for PS. Default: `None`.
- **initializer** (`Initializer`, *optional*) –The initialization strategy for the PS embedding, default is `Uniform(scale=0.01)`.
- **embedding_type** (*str, optional*) –The embedding type, configurable parameters ["PS", "data_parallel"], `"PS"` means initializing PS embedding, `"data_parallel"` means initializing data_parallel embedding, and default is `"PS"`.
- **ev_option** (*EmbeddingVariableOption, optional*) –Properties of the PS embedding, is a EmbeddingVariableOption obj which returned by embedding_variable_option function. Default is `None`.
- **multihot_lens** (*int, optional*) –The param only use when *allow_merge* is enabled, and not support now. Default is `None`.
- **optimizer** (*str, optional*) –The type of optimizer in the train mode for PS embedding, cannot be shared among each PS embedding, and currently only `"Adam"`, `"Ftrl"`, `"SGD"` and `"RMSProp"` are supported, and default is `None`.
- **allow_merge** (*bool, optional*) –Whether to enable merge data_parallel embeddings, currently only be False, and default is `False`.
- **optimizer_param** (*float, optional*) –The "initialize accumulator value" param of optimizer which configured by user, representing the init value of moment accumulator, and default is `None`.
- **mode** (*str, optional*) –Run mode, configurable parameters ["train", "predict", "export"], `"train"` means train mode, `"predict"` means predict mode, `"export"` mean export mode, and default is `"train"`.

Returns

- data_parallel embedding - a dict that contain data_parallel embedding information.
- PS embedding - EmbeddingServiceOut, the embedding init object that contains PS embedding information, which contain five parameters: table_id_dict, es_initializer, es_counter_filter, es_padding_keys, es_completion_keys.
  - table_id_dict (dict): key is PS embedding and value is table_id.
  - es_initializer (dict): key is table_id and value is EsInitializer obj which means PS embedding parameters.
  - es_counter_filter (dict): key is table_id and value is filter option.
  - es_padding_keys (dict): key is table_id and value is padding key.
  - es_completion_keys (dict): key is table_id amd value is completion key.

Raises

- **ValueError** –If "name", "init_vocabulary_size", "embedding_dim", "max_feature_count" are not set.
- **ValueError** –If the types of "name", "init_vocabulary_size", "embedding_dim", and "max_feature_count" do not match.
- **ValueError** –If the value of "init_vocabulary_size", "embedding_dim" and "max_feature_count" is less than or equal to 0, or the value of "init_vocabulary_size" is bigger than 2147483647.
- **ValueError** –If the number of PS embedding exceeds 1024.
- **ValueError** –If the value of "optimizer" not in ["adam", "adagrad", "adamw", "ftrl", "sgd", "rmsprop"].
- **TypeError** –If the type of "initializer" is not EsInitializer obj or not in ["TruncatedNormal", "Uniform", "Constant"].

embedding_table_export(*file_path*, *trainable_var*)

Export Embedding table for each PS embedding and data_parallel embedding.

Note: This function can only be executed by rank 0.

Parameters

- **file_path** (*str*) –The path to export embedding table, and the last character cannot be `"/"`.
- **trainable_var** (*list[parameter]*) –The list of data_parallel embedding parameter.

Returns

The output of EmbeddingTableExport operator and data_parallel embedding export result.

embedding_table_import(*file_path*)

Import embedding file from file path.

Parameters

file_path (*str*) –The path to import embedding table, and the last character cannot be `"/"`.

Returns

The output of EmbeddingTableImport operator and data_parallel embedding import result.

embedding_variable_option(*filter_option=None*, *padding_option=None*, *evict_option=None*, *completion_option=None*, *storage_option=None*, *feature_freezing_option=None*, *communication_option=None*)

Set variable option for PS embedding.

Parameters

- **filter_option** (*CounterFilter, optional*) –The option of counter filter. Default is `None`.
- **padding_option** (*PaddingParamsOption, optional*) –The option of padding key. Default is `None`.
- **evict_option** (*EvictOption, optional*) –The option evict. Default is `None`.
- **completion_option** (*CompletionKeyOption, optional*) –The option of completion key. Default is `None`.
- **storage_option** (*None, optional*) –Reserved option, currently not supported. Default is `None`.
- **feature_freezing_option** (*None, optional*) –Reserved option, currently not supported. Default is `None`.
- **communication_option** (*None, optional*) –Reserved option, currently not supported. Default is `None`.

Returns

EmbeddingVariableOption object, used as the ev_option parameter for *mindspore.experimental.es.EmbeddingService.embedding_init()*.

Raises

- **TypeError** –If value of "filter_option" is not None and the type of "filter_option" is not CounterFilter.
- **TypeError** –If value of "padding_option" is not None and the type of "padding_option" is not PaddingParamsOption.
- **TypeError** –If value of "completion_option" is not None and the type of "completion_option" is not CompletionKeyOption.
- **TypeError** –If value of "evict_option" is not None and the type of "evict_option" is not EvictOption.

evict_option(*steps_to_live*)

Set evict option for each PS embedding.

Parameters

steps_to_live (*int*) –The steps set for evict key.

Returns

EvictOption object.

Raises

- **TypeError** –If the type of "steps_to_live" is not int.
- **ValueError** –If the value of "steps_to_live" is not greater than 0.

incremental_embedding_table_export(*file_path*)

Incremental export embedding table for each PS embedding.

Note: This function can only be executed by rank 0.

Parameters

file_path (*str*) –The path to incremental export embedding table, and the last character cannot be `"/"`.

Returns

The output of EmbeddingTableExport op.

init_table()

Init table for data_parallel embedding.

Returns

A dict of data_parallel embedding parameter, that key is data_parallel embedding name, value is data_parallel embedding parameter.

padding_param(*padding_key*, *mask=True*, *mask_zero=False*)

Init padding key option for each PS embedding.

Parameters

- **padding_key** (*int*) –The value for padding key, must be a genuine and legal hash key.
- **mask** (*bool, optional*) –Whether to update padding key. If set to false, it will not be updated. Default is `True`.

- **mask_zero** (*bool, optional*) –Whether to update padding key when key is 0. Default is `False`.

Returns

PaddingParamsOption object.

Raises

- **TypeError** –If the type of "padding_key" is not int.
- **TypeError** –If the type of "mask" is not bool.

17.2.2 mindspore.experimental.es.EsEmbeddingLookup

class mindspore.experimental.es.**EsEmbeddingLookup**(*table_id*, *es_initializer*, *embedding_dim*, *max_key_num*, *optimizer_mode=None*, *optimizer_params=None*, *es_filter=None*, *es_padding_key=None*, *es_completion_key=None*)

Look up a PS embedding.

Warning: This is an experimental EmbeddingService API that is subject to change.

Parameters

- **table_id** (*int*) –The table id.
- **es_initializer** (*EsInitializer*) –The EsInitialize object for PS embedding with table_id, which can be None when the inference is performed.
- **embedding_dim** (*int*) –The embedding dim of keys for PS embedding with table_id.
- **max_key_num** (*int*) –The num of keys when lookup.
- **optimizer_mode** (*str*) –The type of optimizer. Default is `None`.
- **optimizer_params** (*tuple[float]*) –The parameters of optimizer. Default is `None`.
- **es_filter** (*CounterFilter*) –The option of counter filter for PS embedding with table_id. Default is `None`.
- **es_padding_key** (*PaddingParamsOption*) –The option of padding key for PS embedding with table_id. Default is `None`.
- **es_completion_key** (*CompletionKeyOption*) –The option of completion key for PS embedding with table_id. Default is `None`.

Inputs:

- **keys** (Tensor): The keys of each feature in PS embedding.
- **actual_keys_input** (Tensor): Tensor composed of all unique elements of keys.
- **unique_indices** (Tensor): The index value of each element in keys to actual_keys_input .
- **key_count** (Tensor): The count of each element in the actual_keys_input to keys.

Supported Platforms:

`Atlas A2 training series products`

17.2.3 mindspore.experimental.es.ESEmbeddingSmallTableLookup

class mindspore.experimental.es.**ESEmbeddingSmallTableLookup**(*name*, *rank_id*, *rank_size*, *small_table_to_variable*)

Look up a data_parallel embedding.

> **Warning:** This is an experimental EmbeddingService API that is subject to change.

Parameters

- **name** (*str*) –The data_parallel embedding name.
- **rank_id** (*int*) –The rank id when look up data_parallel embedding key.
- **rank_size** (*int*) –The rank size when look up data_parallel embedding key.
- **small_table_to_variable** (*dict[str, parameter]*) –The dict to restore data_parallel embedding information: key is table name, value is parameter.

Inputs:

- **ids_list** (Tensor) - The keys of each feature in data_parallel embedding.

Supported Platforms:

`Atlas A2 training series products`

CHAPTER

EIGHTEEN

MINDSPORE.OPS.PRIMITIVE

operators that can be used for constructor function of Cell

For more information about dynamic shape support status, please refer to Dynamic Shape Support Status of primitive Interface .

For more information about the support for the bfloat16 data type, please refer to Support List .

For the details about the usage constraints of each operator in the operator parallel process, refer to Usage Constraints During Operator Parallel .

The module import method is as follows:

```
import mindspore.ops as ops
```

Compared with the previous version, the added, deleted and supported platforms change information of *mindspore.ops.primitive* operators in MindSpore, please refer to the link mindspore.ops.primitive API Interface Change .

18.1 Operator Primitives

mindspore.ops.Primitive	Primitive is the base class of operator primitives in python.
mindspore.ops.PrimitiveWithCheck	PrimitiveWithCheck is the base class of primitives in python, which defines functions to check the input arguments of operators, but uses the infer method registered in c++ source codes.
mindspore.ops.PrimitiveWithInfer	PrimitiveWithInfer is the base class of primitives in python and defines functions for tracking inference in python.

18.1.1 mindspore.ops.Primitive

class mindspore.ops.**Primitive**(*name*)

Primitive is the base class of operator primitives in python.

Parameters

name (*str*) – Name for the current Primitive.

Examples

```
>>> from mindspore.ops import prim_attr_register, Primitive
>>> add = Primitive('add')
>>>
>>> # or work with prim_attr_register:
>>> # init a Primitive class with attr1 and attr2
>>> class Add(Primitive):
...     @prim_attr_register
...     def __init__(self, attr1, attr2):
...         '''init for add'''
...     # check attr1 and attr2 or do some initializations
...     # init a Primitive obj with attr1=1 and attr2=2
>>> add = Add(attr1=1, attr2=2)
```

add_prim_attr(*name*, *value*)

Add primitive attribute.

Parameters

- **name** (*str*) –Attribute Name.
- **value** (*Any*) –Attribute value.

Examples

```
>>> from mindspore import ops
>>> a = ops.Add()
>>> a = a.add_prim_attr("attr",1)
>>> out = a.attrs["attr"]
>>> print(out)
1
```

check_elim(**args*)

Check if the primitive can be eliminated. Subclass in need should override this method.

Parameters

args (*Primitive args*) –Same as arguments of current Primitive.

Returns

A tuple consisting of two elements. The first element means if the primitive can be calculated in compiling stage, the second element is calculated result.

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.ops import prim_attr_register, Primitive
>>> class AddN(Primitive):
...     @prim_attr_register
...     def __init__(self):
...         self.init_prim_io_names(inputs=["inputs"], outputs=["sum"])
...     def check_elim(self, inputs):
```

(continues on next page)

(continued from previous page)

```
...         if len(inputs) != 1:
...             return (False, None)
...         if isinstance(inputs[0], Tensor):
...             return (True, inputs[0])
...
>>> addn = AddN()
>>> input_x = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> output = addn.check_elim((input_x,))
>>> print(output)
(True, Tensor(shape=[3], dtype=Float32, value= [ 1.00000000e+00,  2.00000000e+00,  3.
↪00000000e+00]))
```

del_prim_attr(*name*)

Delete primitive attribute.

Parameters

name (*str*) – Attribute Name.

Examples

```
>>> from mindspore import ops
>>> a = ops.Add()
>>> a = a.add_prim_attr("attr",1)
>>> a = a.del_prim_attr("attr")
>>> print(a.attrs)
{}
```

init_prim_io_names(*inputs*, *outputs*)

Initialize the name of inputs and outputs of Tensor or attributes.

Parameters

- **inputs** (*list[str]*) – list of inputs names.
- **outputs** (*list[str]*) – list of outputs names.

Examples

```
>>> from mindspore import ops
>>> a = ops.Add()
>>> a.init_prim_io_names(["x","y"],["sum"])
>>> print(a.input_names)
['x','y']
>>> print(a.output_names)
['sum']
```

place(*role*, *rank_id*)

Set the label for this primitive. This label tells MindSpore compiler on which process this operator should be launched. And each process's identical label consists of input 'role' and 'rank_id'. So by setting different operators with different labels, which will be launched on different processes, users can launch a distributed training job.

Note:

- This method is effective only after "mindspore.communication.init()" is called for dynamic cluster building.

Parameters

- **role** (*str*) –The role of the process on which this operator will be launched. Only 'MS_WORKER' is supported for now.
- **rank_id** (*int*) –The rank id of the process on which this operator will be launched. The rank_id is unique in processes with the same role.

Examples

```
>>> from mindspore import context
>>> from mindspore import ops
>>> context.set_context(mode=context.GRAPH_MODE)
>>> matmul = ops.MatMul()
>>> matmul.place('MS_WORKER', 0)
```

recompute(*mode=True*)

Set the primitive recomputed. If a primitive set recomputed feeds into some backward nodes for computing gradient, rather than storing the intermediate activation computed in forward pass, we will recompute it in backward pass.

Note:

- If the computation involves something like randomization or global variable, the equivalence is not guaranteed currently.
- Not supported in pynative mode

Parameters

mode (*bool*) –Specifies whether the primitive is recomputed. Default: `True` .

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor, ops, nn
>>> class NetRecompute(nn.Cell):
...     def __init__(self):
...         super(NetRecompute,self).__init__()
...         self.relu = ops.ReLU().recompute()
...         self.sqrt = ops.Sqrt()
...     def construct(self, x):
...         out = self.relu(x)
...         return self.sqrt(out)
...
>>> class GradNet(nn.Cell):
...     def __init__(self, network):
...         super(GradNet,self).__init__()
...         self.network = network
...         self.grad = ops.GradOperation()
...     def construct(self, x):
...         g_out = self.grad(self.network)(x)
...         return g_out
```

(continues on next page)

(continued from previous page)

```
...
>>> x = Tensor(np.array([-1,1]).astype(np.float32))
>>> net = NetRecompute()
>>> grad = GradNet(net)
>>> a = grad(x)
>>> print(a)
[0. 0.5]
```

set_device(*device_target*)

Set primitive been executed device.

Parameters

device_target (*str*) – The target device to run, support "Ascend", "GPU", and "CPU".

Examples

```
>>> from mindspore import ops
>>> a = ops.Add()
>>> a = a.set_device("GPU")
>>> print(a.primitive_target)
GPU
```

set_prim_instance_name(*instance_name*)

Set instance name to primitive operator.

Note: It will be called by default when user defines primitive operator.

Parameters

instance_name (*str*) – Instance name of primitive operator set by user.

Examples

```
>>> from mindspore import ops
>>> a = ops.Add()
>>> a = a.set_prim_instance_name("add")
>>> print(a.instance_name)
add
```

set_stage(*stage*)

Add stage id to primitive attribute.

Note: It is valid only in semi auto parallel. In other parallel modes, please set it to be 0.

Parameters

stage (*int*) – The stage id for the current operation.

Examples

```
>>> from mindspore import ops
>>> add = ops.Add()
>>> print(add.set_stage(0))
Prim[Add]<stage=0>
```

shard(*in_strategy=None*, *out_strategy=None*)
Add strategies to primitive attribute.

Note: It is valid only in semi auto parallel or auto parallel mode. In other parallel modes, strategies set here will be ignored.

Parameters

- **in_strategy** (*tuple*) –Describe the split strategy of operator input. Default: `None` .
- **out_strategy** (*tuple*) –Describe the split strategy of operator output, it is only for certain operators, such as MatMul. Default: `None` .

Examples

```
>>> from mindspore import ops
>>> add = ops.Add()
>>> print(add.shard(((1, 1), (1, 1))))
Prim[Add]<in_strategy=((1, 1), (1, 1)), out_strategy=None>
>>> # using layout
>>> from mindspore import Layout
>>> layout = Layout((2, 2, 2), ("dp", "sp", "mp"))
>>> layout_tuple = (layout("dp", "sp"), layout("sp", "mp"))
>>> from mindspore import ops
>>> matmul = ops.MatMul()
>>> print(matmul.shard(layout_tuple))
Prim[MatMul]<in_layout=({'device_matrix': (2, 2, 2), 'tensor_map': (2, 1)},
{'device_matrix': (2, 2, 2), 'tensor_map': (1, 0)})>
>>> # using layout with None
>>> from mindspore import Layout
>>> layout = Layout((2, 2, 2), ("dp", "sp", "mp"))
>>> layout_tuple = (layout("dp", "sp"), layout("sp", "None")) # "None" means the axis␣
↪would not be split
>>> from mindspore import ops
>>> matmul = ops.MatMul()
>>> print(matmul.shard(layout_tuple))
Prim[MatMul]<in_layout=({'device_matrix': (2, 2, 2), 'tensor_map': (2, 1)},
{'device_matrix': (2, 2, 2), 'tensor_map': (1, -1)})>
```

property update_parameter
Return whether the primitive will update the value of parameter.

18.1.2 mindspore.ops.PrimitiveWithCheck

class mindspore.ops.**PrimitiveWithCheck**(*name*)

PrimitiveWithCheck is the base class of primitives in python, which defines functions to check the input arguments of operators, but uses the infer method registered in c++ source codes.

There are three methods can be overridden to define the check logic of the primitive: __check__(), check_shape(), check_dtype(). If __check__() is defined in primitive, the __check__() has the highest priority to be called. If __check__() is not defined, check_shape() and check_dtype() can be defined to describe the check logic of the shape and type. Method infer_value() can also be defined (such as PrimitiveWithInfer) for constant propagation.

More on how to customize a Op, please refer to Custom Operators.

Parameters

name (*str*) – Name of the current Primitive.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore.ops import prim_attr_register, PrimitiveWithCheck
>>> # init a Primitive class with check
>>> class Flatten(PrimitiveWithCheck):
...     @prim_attr_register
...     def __init__(self):
...         pass
...     def check_shape(self, input_x):
...         Validator.check_int(len(input_x), 1, validator.GE, 'input_x rank', self.name)
...
...     def check_dtype(self, input_x):
...         Validator.check_subclass("input_x", input_x, mstype.tensor_type, self.name)
...
>>> # init a Primitive obj
>>> add = Flatten()
```

check_dtype(**args*)

Check data types of input args.

Parameters

args (*mindspore.dtype*) – data type of inputs.

Returns

None.

check_shape(**args*)

Check shapes of input args.

Note: The shape of scalar is an empty tuple.

Parameters

args (*tuple(int)*) – shapes of input tensors.

Returns

None.

18.1.3 mindspore.ops.PrimitiveWithInfer

class mindspore.ops.**PrimitiveWithInfer**(*name*)

PrimitiveWithInfer is the base class of primitives in python and defines functions for tracking inference in python.

There are four method can be overridden to define the infer logic of the primitive: __infer__(), infer_shape(), infer_dtype(), and infer_value(). If __infer__() is defined in primitive, the __infer__() has the highest priority to be called. If __infer__() is not defined, infer_shape() and infer_dtype() can be defined to describe the infer logic of the shape and type. The infer_value() is used for constant propagation.

More on how to customize a Op, please refer to Custom Operators.

Parameters

name (*str*) – Name of the current Primitive.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore.ops import prim_attr_register, PrimitiveWithInfer
>>> # init a Primitive class with infer
>>> class Add(PrimitiveWithInfer):
...     @prim_attr_register
...     def __init__(self):
...         pass
...
...     def infer_shape(self, x, y):
...         return x # output shape same as first input 'x'
...
...     def infer_dtype(self, x, y):
...         return x # output type same as first input 'x'
...
>>> # init a Primitive obj
>>> add = Add()
```

18.2 Neural Network Layer Operators

18.2.1 Neural Network

API Name	Description	Supported Platforms
mindspore.ops.AvgPool	Average pooling operation.	Ascend GPU CPU
mindspore.ops.AvgPool3D	3D Average pooling operation.	Ascend GPU CPU
mindspore.ops.BatchNorm	Batch Normalization for input data and updated parameters.	Ascend GPU CPU

continues on next page

Table 2 – continued from previous page

mindspore.ops.Conv2D	2D convolution layer.	Ascend GPU CPU
mindspore.ops.Conv2DTranspose	Calculates a 2D transposed convolution, which can be regarded as Conv2d for the gradient of the input, also called deconvolution, although it is not an actual deconvolution.	Ascend GPU CPU
mindspore.ops.Conv3D	3D convolution layer.	Ascend GPU CPU
mindspore.ops.Conv3DTranspose	Computes a 3D transposed convolution, which is also known as a deconvolution (although it is not an actual deconvolution).	Ascend GPU CPU
mindspore.ops.CTCGreedyDecoder	Performs greedy decoding on the logits given in inputs.	Ascend GPU CPU
mindspore.ops.Dense	Refer to mindspore.ops.dense() for more details.	Ascend GPU CPU
mindspore.ops.Dropout	During training, randomly zeroes some of the elements of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution.	Ascend GPU CPU
mindspore.ops.Dropout2D	During training, randomly zeroes some channels of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution(For a 4-dimensional tensor with a shape of (N, C, H, W), the channel feature map refers to a 2-dimensional feature map with the shape of (H, W)).	Ascend GPU CPU
mindspore.ops.Dropout3D	During training, randomly zeroes some channels of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution(For a 5-dimensional tensor with a shape of NCDHW, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).	Ascend GPU CPU
mindspore.ops.DynamicGRUV2	Applies a single-layer gated recurrent unit (GRU) to an input sequence.	Ascend
mindspore.ops.DynamicRNN	Applies a recurrent neural network to the input.	Ascend
mindspore.ops.Flatten	Flattens a tensor without changing its batch size on the 0-th axis.	Ascend GPU CPU
mindspore.ops.FractionalMaxPool3DWithFixedKsize	Applies a 3D fractional max pooling to an input signal composed of multiple input planes.	Ascend GPU CPU
mindspore.ops.GridSampler2D	This operation samples 2d *input_x* by using interpolation based on flow field grid, which is usually gennerated by mindspore.ops.affine_grid().	Ascend GPU CPU
mindspore.ops.GridSampler3D	Given an input and a grid, the output is calculated using the input values and pixel positions in the grid.	Ascend GPU CPU
mindspore.ops.LayerNorm	Applies the Layer Normalization to the input tensor.	Ascend GPU CPU
mindspore.ops.LRN	Local Response Normalization.	GPU CPU
mindspore.ops.LSTM	Performs the Long Short-Term Memory (LSTM) on the input.	GPU CPU
mindspore.ops.MaxPool	Max pooling operation.	Ascend GPU CPU
mindspore.ops.MaxPool3D	Applies a 3D max pooling over an input Tensor which can be regarded as a composition of 3D planes.	Ascend GPU CPU

continues on next page

Table 2 – continued from previous page

mindspore.ops.MaxPool3DWithArgmax	Performs a 3D max pooling on the input Tensor and returns both max values and indices.	Ascend GPU CPU
mindspore.ops.MaxUnpool2D	Calculates the partial inverse of MaxPool2D operation.	Ascend GPU CPU
mindspore.ops.MaxUnpool3D	Computes the inverse of *mindspore.ops.MaxPool3D*.	Ascend GPU CPU
mindspore.ops.MirrorPad	Pads the input tensor according to the paddings and mode.	Ascend GPU CPU
mindspore.ops.Pad	Pads the input tensor according to the paddings.	Ascend GPU CPU
mindspore.ops.EmbeddingLookup	Returns a slice of input tensor based on the specified indices.	Ascend GPU CPU
mindspore.ops.Padding	Extends the last dimension of the input tensor from 1 to pad_dim_size, by filling with 0.	Ascend GPU CPU
mindspore.ops.ResizeBicubic	Resize images to size using bicubic interpolation.	Ascend GPU CPU
mindspore.ops.ResizeNearestNeighbor	Resizes the input tensor to a given size by using the nearest neighbor algorithm.	Ascend GPU CPU

mindspore.ops.AvgPool

class mindspore.ops.**AvgPool**(*kernel_size=1*, *strides=1*, *pad_mode='VALID'*, *data_format='NCHW'*)

Average pooling operation.

Refer to *mindspore.ops.avg_pool2d()* for more details.

Parameters

- **kernel_size** (*Union[int, tuple[int]], optional*) – The size of kernel used to take the average value, is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively. Default: `1` .
- **strides** (*Union[int, tuple[int]], optional*) – The distance of kernel moving, an int number that represents the height and width of movement are both strides, or a tuple of two int numbers that represent height and width of movement respectively. Default: `1` .
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `"SAME"` or `"VALID"` . Default: `"VALID"` .
  - `"SAME"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded.
- **data_format** (*str, optional*) – The format of input and output data. It should be `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$. Supported dtypes: float16, float32, float64.

Outputs:

Tensor, with shape $(N, C_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** – If *kernel_size* or *strides* is neither int nor tuple.

- **TypeError** – If dtype of *x* is not float16, float32 or float64.
- **ValueError** – If *kernel_size* or *strides* is less than 1.
- **ValueError** – If *pad_mode* is neither 'valid' nor 'same' with not case sensitive.
- **ValueError** – If *data_format* is neither 'NCHW' nor 'NHWC'.
- **ValueError** – If length of shape of *x* is not equal to 4.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, nn
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.avgpool_op = ops.AvgPool(pad_mode='VALID', kernel_size=2, strides=1)
...
...     def construct(self, x):
...         result = self.avgpool_op(x)
...         return result
...
>>> x = Tensor(np.arange(1 * 3 * 3 * 4).reshape(1, 3, 3, 4), mindspore.float32)
>>> net = Net()
>>> output = net(x)
>>> print(output)
[[[[ 2.5   3.5   4.5]
   [ 6.5   7.5   8.5]]
  [[14.5  15.5  16.5]
   [18.5  19.5  20.5]]
  [[26.5  27.5  28.5]
   [30.5  31.5  32.5]]]]
```

mindspore.ops.AvgPool3D

class mindspore.ops.**AvgPool3D**(*kernel_size=1*, *strides=1*, *pad_mode='valid'*, *pad=0*, *ceil_mode=False*, *count_include_pad=True*, *divisor_override=0*, *data_format='NCDHW'*)

3D Average pooling operation.

Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$, AvgPool3D outputs regional average in the (D_{in}, H_{in}, W_{in})-dimension. Given kernel size $ks = (d_{ker}, h_{ker}, w_{ker})$ and stride $s = (s_0, s_1, s_2)$, the operation is as follows:

$$\text{output}(N_i, C_j, d, h, w) = \frac{1}{d_{ker} * h_{ker} * w_{ker}} \sum_{l=0}^{d_{ker}-1} \sum_{m=0}^{h_{ker}-1} \sum_{n=0}^{w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Note: This interface currently does not support Atlas A2 training series products.

Parameters

- **kernel_size** (*Union[int, tuple[int]], optional*) – The size of kernel used to take the average value, is an int number that represents depth, height and width are both kernel_size, or a tuple of three int numbers that represent depth, height and width respectively. Default: `1` . The value range is: [1, 255].
- **strides** (*Union[int, tuple[int]], optional*) – The distance of kernel moving, an int number that represents the depth, height and width of movement are both strides, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: `1` . The value range is: [1, 63].
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *pad* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *pad* parameter. If this mode is set, *pad* must be greater than or equal to 0.
- **pad** (*Union(int, tuple[int], list[int]), optional*) – The pad value to be filled. Default: `0` . If *pad* is an integer, the paddings of head, tail, top, bottom, left and right are the same, equal to pad. If *pad* is a tuple of six integers, the padding of head, tail, top, bottom, left and right equal to pad[0], pad[1], pad[2], pad[3], pad[4] and pad[5] correspondingly.
- **ceil_mode** (*bool, optional*) – If `True` , ceil instead of floor to compute the output shape. Default: `False` .
- **count_include_pad** (*bool, optional*) – If `True` , averaging calculation will include the zero-padding. Default: `True` .
- **divisor_override** (*int, optional*) – If specified, it will be used as divisor in the averaging calculation, otherwise kernel_size will be used. Default: `0` .
- **data_format** (*str, optional*) – The optional value for data format. Currently only support `'NCDHW'` . Default: `'NCDHW'` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$. Currently support float16, float32 and float64 data type.

Outputs:

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$. Has the same data type with *x*.

Raises

- **TypeError** – If *kernel_size*, *strides* or *pad* is neither an int not a tuple.
- **TypeError** – If *ceil_mode* or *count_include_pad* is not a bool.
- **TypeError** – If *pad_mode* or *data_format* is not a string.
- **TypeError** – If *divisor_override* is not an int.
- **ValueError** – If numbers in *kernel_size* or *strides* are not positive.
- **ValueError** – If *kernel_size* or *strides* is a tuple whose length is not equal to 3.
- **ValueError** – If *pad_mode* is not one of 'same', 'valid' or 'pad'.
- **ValueError** – If *pad* is a tuple whose length is not equal to 6.
- **ValueError** – If element of *pad* is less than 0.

- **ValueError** – If *pad_mode* is not equal to 'pad' and *pad* is not equal to 0 or (0, 0, 0, 0, 0, 0).
- **ValueError** – If *data_format* is not 'NCDHW'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> x = Tensor(np.arange(1 * 2 * 2 * 2 * 3).reshape((1, 2, 2, 2, 3)), mindspore.float16)
>>> avg_pool3d = ops.AvgPool3D(kernel_size=2, strides=1, pad_mode="valid")
>>> output = avg_pool3d(x)
>>> print(output)
[[[[[ 5.  6.]]]
  [[[17. 18.]]]]]
```

mindspore.ops.BatchNorm

class mindspore.ops.**BatchNorm**(*is_training=False*, *epsilon=1e-5*, *momentum=0.1*, *data_format='NCHW'*)

Batch Normalization for input data and updated parameters.

Batch Normalization is widely used in convolutional neural networks. This operation applies Batch Normalization over inputs to avoid internal covariate shift as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. It rescales and recenters the features using a mini-batch of data and the learned parameters can be described in the following formula,

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

where γ is scale, β is bias, ϵ is epsilon, $mean$ is the mean of x, $variance$ is the variance of x.

Warning:

- If the operation is used for inference, and outputs "reserve_space_1" and "reserve_space_2" are available, then "reserve_space_1" has the same value as "mean" and "reserve_space_2" has the same value as "variance".
- For Ascend 310, the result accuracy fails to reach 1‰ due to the square root instruction.

Parameters

- **is_training** (*bool, optional*) – If *is_training* is `True` , *mean* and *variance* are computed during training. If *is_training* is `False` , they're loaded from checkpoint during inference. Default: `False` .
- **epsilon** (*float, optional*) – A small value added for numerical stability. Default: `1e-5`, value must be (0, 1] .
- **momentum** (*float, optional*) – The hyper parameter to compute moving average for running_mean and running_var (e.g. $new\_running\_mean = (1 - momentum) * running\_mean + momentum * current\_mean$). Momentum value must be [0, 1]. Default: `0.1` .
- **data_format** (*str, optional*) – The optional value for data format, is `'NHWC'` or `'NCHW'`, and the `'NHWC'` format is only supported in GPU target. Default: `"NCHW"` .

Inputs:

- **input_x** (Tensor) - Tensor of shape (N, C), with float16 or float32 data type.
- **scale** (Union[Parameter, Tensor]) - Tensor or Parameter of shape $(C,)$, with float16 or float32 data type.
- **bias** (Union[Parameter, Tensor]) - Tensor or Parameter of shape $(C,)$, has the same data type with *scale*.
- **mean** (Union[Parameter, Tensor]) - Tensor or Parameter of shape $(C,)$, has the same data type with *scale*.
- **variance** (Union[Parameter, Tensor]) - Tensor or Parameter of shape $(C,)$, has the same data type with *scale*.

Outputs:

Tuple of 5 Tensors, the normalized inputs and the updated parameters.

- **output_x** (Tensor) - The same type and shape as the input_x. The shape is (N, C).
- **batch_mean** (Tensor) - The mean calculated per-dimension over the mini-batches, shape is $(C,)$.
- **batch_variance** (Tensor) - The variance calculated per-dimension over the mini-batches, shape is $(C,)$.
- **reserve_space_1** (Tensor) - The mean that needs to be reused when calculating gradients, one-dimensional Tensor. The shape is $(C,)$.
- **reserve_space_2** (Tensor) - The variance that needs to be reused when calculating gradients, one-dimensional Tensor. The shape is $(C,)$.

Raises

- **TypeError** –If *is_training* is not a bool.
- **TypeError** –If dtype of *epsilon* or *momentum* is not float.
- **TypeError** –If *data_format* is not a str.
- **TypeError** –If *input_x*, *scale*, *bias*, *mean* or *variance* is not a Tensor.
- **TypeError** –If dtype of *input_x*, *scale* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones([2, 2]), mindspore.float32)
>>> scale = Tensor(np.ones([2]), mindspore.float32)
>>> bias = Tensor(np.ones([2]), mindspore.float32)
>>> mean = Tensor(np.ones([2]), mindspore.float32)
>>> variance = Tensor(np.ones([2]), mindspore.float32)
>>> batch_norm = ops.BatchNorm()
>>> output = batch_norm(input_x, scale, bias, mean, variance)
>>> print(output[0])
[[1. 1.]
 [1. 1.]]
```

mindspore.ops.Conv2D

class mindspore.ops.**Conv2D**(*out_channel*, *kernel_size*, *mode=1*, *pad_mode='valid'*, *pad=0*, *stride=1*, *dilation=1*, *group=1*, *data_format='NCHW'*)

2D convolution layer.

Applies a 2D convolution over an input tensor which is typically of shape $(N, C_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, H is feature height, W is feature width.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where $bias$ is the output channel bias, $ccor$ is the cross-correlation, , $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$ represents the slice of the j-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1]), where kernel_size[0] and kernel_size[1] are the height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1])$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Note: On Ascend platform, only group convolution in depthwise convolution scenarios is supported. That is, when *group>1*, condition *in_channels* = *out_channels* = *group* must be satisfied.

Parameters

- **out_channel** (*int*) –Specifies output channel C_{out}.
- **kernel_size** (*Union[int, tuple[int]]*) –Specifies the height and width of the 2D convolution kernel. It can be a single int or a tuple of 2 integers. A single int means the value is for both the height and the width. A tuple of 2 ints means the first value is for the height and the other is for the width.
- **mode** (*int, optional*) –Modes for different convolutions. The value is currently not used. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *pad* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad* must be 0.

  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *pad* parameter. If this mode is set, *pad* must be greater than or equal to 0.

- **pad** (*Union(int, tuple[int]), optional*) – Specifies the amount of padding to apply on input when *pad_mode* is set to `"pad"`. It can be a single int or a tuple of 4 ints. If *pad* is one integer, the paddings of top, bottom, left and right are the same, equal to *pad*. If *pad* is a tuple with four integers, the paddings of top, bottom, left and right will be equal to pad[0], pad[1], pad[2], and pad[3] accordingly. Default: `0` .
- **stride** (*Union(int, tuple[int]), optional*) – Specifies the stride of the convolution kernel's movement. It can be a single int or a tuple of two or four ints. A single int means the stride is the same in both the height and width directions. A tuple of two ints indicates the strides in the height and width directions, respectively. For a tuple of four ints, the two ints correspond to (N, C) dimension are treated as 1, and the two correspond to (H, W) dimensions is the step size in the height and width directions respectively. Default: `1` .
- **dilation** (*Union(int, tuple[int]), optional*) – Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 2 or 4 integers. A single int means the dilation size is the same in both the height and width directions. A tuple of two ints represents the dilation size in the height and width directions, respectively. For a tuple of four ints, the two ints correspond to (N, C) dimension are treated as 1, and the two correspond to (H, W) dimensions is the dilation size in the height and width directions respectively. Assuming $dilation = (d0, d1)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the height direction and $d1 - 1$ elements in the width direction. The values in the height and width dimensions are in the ranges [1, H] and [1, W], respectively. Default: `1` .
- **group** (*int, optional*) – Specifies the number of groups dividing *x*'s input channel when applying group convolution. Default: `1` .
- **data_format** (*str, optional*) – The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `"NCHW"`. (NHWC is only supported in GPU now.)

Inputs:

- **x** (Tensor) - Input tensor of shape $(N, C_{in}, H_{in}, W_{in})$ or $(N, H_{in}, W_{in}, C_{in})$ depending on *data_format* .
- **weight** (Tensor) - The convolutional kernel value, it should has shape $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1])$.

Outputs:

Tensor, the value that applied 2D convolution. The shape is $(N, C_{out}, H_{out}, W_{out})$ or $(N, H_{out}, W_{out}, C_{out})$. To see how different pad modes affect the output shape, please refer to *mindspore.nn.Conv2d* for more details.

Raises

- **TypeError** – If *kernel_size*, *stride*, *pad* or *dilation* is neither an int nor a tuple.
- **TypeError** – If *out_channel* or *group* is not an int.
- **ValueError** – If *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *pad_mode* is not one of `'same'`, `'valid'` or `'pad'`.
- **ValueError** – If *pad* is a tuple whose length is not equal to 4.
- **ValueError** – If *pad_mode* it not equal to `'pad'` and *pad* is not equal to `(0, 0, 0, 0)`.
- **ValueError** – If *data_format* is neither `'NHWC'` nor `'NCHW'` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: All parameters use default values.
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3)
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 30, 30)
>>> # case 2: pad_mode="pad", other parameters being default.
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3, pad_mode="pad", pad=(4, 10, 4, 10))
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 44, 44)
>>> # case 3: stride=(2, 4), other parameters being default.
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3, stride=(2, 4))
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 15, 8)
>>> # case 4: dilation=2, other parameters being default.
>>> x = Tensor(np.ones([10, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3, dilation=2)
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 28, 28)
>>> # case 5: group=2, other parameters being default.
>>> x = Tensor(np.ones([10, 64, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3, group=2)
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 30, 30)
>>> # case 6: All parameters are specified.
>>> x = Tensor(np.ones([10, 64, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> conv2d = ops.Conv2D(out_channel=32, kernel_size=3, pad_mode="pad",
...                     pad=(4, 10, 4, 10), stride=(2, 4), dilation=2,  group=2)
>>> output = conv2d(x, weight)
>>> print(output.shape)
(10, 32, 21, 11)
```

mindspore.ops.Conv2DTranspose

class mindspore.ops.**Conv2DTranspose**(*out_channel*, *kernel_size*, *pad_mode='valid'*, *pad=0*, *pad_list=None*, *mode=1*, *stride=1*, *dilation=1*, *group=1*, *data_format='NCHW'*)

Calculates a 2D transposed convolution, which can be regarded as Conv2d for the gradient of the input, also called deconvolution, although it is not an actual deconvolution. Because it cannot restore the original input data completely, but it can restore the shape of the original input.

Parameters

- **out_channel** (*int*) –The dimensionality of the output space.
- **kernel_size** (*Union[int, tuple[int]]*) –The size of the convolution window.
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side. If this mode is set, *pad* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the height and width directions is determined by the *pad* parameter. If this mode is set, *pad* must be greater than or equal to 0.

  Please refer to *mindspore.nn.Conv2dTranspose* for more specifications about *pad_mode*.
- **pad** (*Union[int, tuple[int]], optional*) –The pad value to be filled. Default: `0` . If *pad* is an integer, the paddings of top, bottom, left and right are the same, equal to pad. If *pad* is a tuple of four integers, the padding of top, bottom, left and right equal to pad[0], pad[1], pad[2], and pad[3] correspondingly.
- **pad_list** (*Union[str, None], optional*) –The pad list like (top, bottom, left, right). Default: `None` .
- **mode** (*int, optional*) –Modes for different convolutions. The value is currently not used. Default: `1` .
- **stride** (*Union[int, tuple[int]], optional*) –The stride to be applied to the convolution filter. Default: `1` .
- **dilation** (*Union[int, tuple[int]], optional*) –Specifies the dilation rate to be used for the dilated convolution. Default: `1` .
- **group** (*int, optional*) –Splits input into groups. Default: `1` .
- **data_format** (*str, optional*) –The format of input and output data. It should be `'NHWC'` or `'NCHW'` . Default is `'NCHW'` .

Inputs:

- **dout** (Tensor) - the gradients with respect to the output of the convolution. The shape conforms to the default data_format $(N, C_{out}, H_{out}, W_{out})$.
- **weight** (Tensor) - Set size of kernel is (K_1, K_2), then the shape is $(C_{out}, C_{in}, K_1, K_2)$.
- **input_size** (Tensor) - A tuple describes the shape of the input which conforms to the format $(N, C_{in}, H_{in}, W_{in})$.

Outputs:

Tensor, the gradients with respect to the input of convolution. It has the same shape as the input.

Raises

- **TypeError** –If *kernel_size*, *stride*, *pad* or *dilation* is neither an int nor a tuple.

- **TypeError** – If *out_channel* or *group* is not an int.
- **ValueError** – If *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *pad_mode* is not one of `'same'`, `'valid'` or `'pad'`.
- **ValueError** – If *padding* is a tuple whose length is not equal to 4.
- **ValueError** – If *pad_mode* it not equal to `'pad'` and *pad* is not equal to (0, 0, 0, 0).
- **ValueError** – If *data_format* is neither `'NCHW'` nor `'NHWC'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> dout = Tensor(np.ones([10, 32, 30, 30]), mindspore.float32)
>>> weight = Tensor(np.ones([32, 32, 3, 3]), mindspore.float32)
>>> x = Tensor(np.ones([10, 32, 32, 32]))
>>> conv2d_transpose_input = ops.Conv2DTranspose(out_channel=32, kernel_size=3)
>>> output = conv2d_transpose_input(dout, weight, ops.shape(x))
>>> print(output.shape)
(10, 32, 32, 32)
```

mindspore.ops.Conv3D

class mindspore.ops.**Conv3D**(*out_channel*, *kernel_size*, *mode=1*, *pad_mode='valid'*, *pad=0*, *stride=1*, *dilation=1*, *group=1*, *data_format='NCDHW'*)

3D convolution layer.

Applies a 3D convolution over an input tensor which is typically of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$, where N is batch size, C is channel number, D, H, W are the depth, height and width of the feature map, respectively.

The output is calculated based on formula:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{ccor}(\text{weight}(C_{\text{out}_j}, k), \text{X}(N_i, k))$$

where *bias* is the output channel bias, $ccor$ is the cross-correlation, $weight$ is the convolution kernel value and X represents the input feature map.

Here are the indices' meanings:

- i corresponds to the batch number, the range is $[0, N-1]$, where N is the batch size of the input.
- j corresponds to the output channel, the range is $[0, C_{out}-1]$, where C_{out} is the number of output channels, which is also equal to the number of kernels.
- k corresponds to the input channel, the range is $[0, C_{in}-1]$, where C_{in} is the number of input channels, which is also equal to the number of channels in the convolutional kernels.

Therefore, in the above formula, $bias(C_{\text{out}_j})$ represents the bias of the j-th output channel, $weight(C_{\text{out}_j}, k)$-th convolutional kernel in the k-th channel, and $X(N_i, k)$ represents the slice of the k-th input channel in the i-th batch of the input feature map.

The shape of the convolutional kernel is given by (kernel_size[0], kernel_size[1], kernel_size[2]) where kernel_size[0], kernel_size[1] and kernel_size[2] are the depth, height and width of the kernel, respectively. If we consider the input and output channels as well as the *group* parameter, the complete kernel shape will be $(C_{out}, C_{in}/\text{group}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$, where *group* is the number of groups dividing *x*'s input channel when applying group convolution.

For more details about convolution layer, please refer to Gradient Based Learning Applied to Document Recognition.

Note:

1. On Ascend platform, $groups = 1$ must be satisfied.
2. On Ascend $dilation$ on depth only supports the case of 1.

Parameters

- **out_channel** (*int*) −Specifies output channel C_{out}.
- **kernel_size** (*Union[int, tuple[int]]*) −Specifies the depth, height and width of the 3D convolution kernel. It can be a single int or a tuple of 3 integers. A single int means the value is for depth, height and the width. A tuple of 3 ints means the first value is for depth and the rest is for the height and width.
- **mode** (*int, optional*) −Modes for different convolutions. It is currently not used. Default: `1` .
- **stride** (*Union[int, tuple[int]], optional*) −The distance of kernel moving, it can be an int number that represents the depth, height and width of movement or a tuple of three int numbers that represent depth, height and width movement respectively. Default: `1` .
- **pad_mode** (*str, optional*) −Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *pad* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *pad* parameter. If this mode is set, *pad* must be greater than or equal to 0.
- **pad** (*Union(int, tuple[int]), optional*) −Specifies the amount of padding to apply on input when *pad_mode* is set to `"pad"`. It can be a single int or a tuple of 6 ints. If *pad* is one integer, the paddings of head, tail, top, bottom, left and right are the same, equal to *pad*. If *pad* is a tuple with 6 integers, the paddings of head, tail, top, bottom, left and right is equal to pad[0], pad[1], pad[2], pad[3], pad[4] and pad[5] accordingly. Default: `0` .
- **dilation** (*Union[int, tuple[int]], optional*) −Specifies the dilation rate to use for dilated convolution. It can be a single int or a tuple of 3 integers. A single int means the dilation size is the same in the depth, height and width directions. A tuple of 3 ints represents the dilation size in the depth, height and width directions, respectively. Assuming $dilation = (d0, d1, d2)$, the convolutional kernel samples the input with a spacing of $d0 - 1$ elements in the depth direction, $d1 - 1$ elements in the height direction, $d2 - 1$ elements in the width direction respectively. The values in the depth, height and width dimensions are in the ranges [1, D], [1, H] and [1, W], respectively. Default: `1` .
- **group** (*int, optional*) −The number of groups into which the filter is divided. *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **data_format** (*str, optional*) −The optional value for data format. Currently only support `"NCDHW"` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, D_{in}, H_{in}, W_{in})$. Currently input data type only support float16 and float32.
- **weight** (Tensor) - Set size of kernel is (k_d, K_h, K_w), then the shape is $(C_{out}, C_{in}/groups, k_d, K_h, K_w)$. Currently weight data type only support float16 and float32.
- **bias** (Tensor) - Tensor of shape (C_{out}). When bias is None, zeros will be used. Default: `None` .

Outputs:

Tensor, the value that applied 3D convolution. The shape is $(N, C_{out}, D_{out}, H_{out}, W_{out})$. To see how different pad modes affect the output shape, please refer to *mindspore.nn.Conv3d* for more details.

Raises

- **TypeError** –If *out_channel* or *group* is not an int.
- **TypeError** –If *kernel_size*, *stride*, *pad* or *dilation* is neither an int nor a tuple.
- **ValueError** –If *out_channel*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** –If *pad* is less than 0.
- **ValueError** –If *pad_mode* is not one of 'same', 'valid' or 'pad'.
- **ValueError** –If *pad* is a tuple whose length is not equal to 6.
- **ValueError** –If *pad_mode* is not equal to 'pad' and *pad* is not equal to (0, 0, 0, 0, 0, 0).
- **ValueError** –If *data_format* is not 'NCDHW'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: specify kernel_size with tuple, all parameters use default values.
>>> x = Tensor(np.ones([16, 3, 10, 32, 32]), mindspore.float16)
>>> weight = Tensor(np.ones([32, 3, 4, 3, 3]), mindspore.float16)
>>> conv3d = ops.Conv3D(out_channel=32, kernel_size=(4, 3, 3))
>>> output = conv3d(x, weight)
>>> print(output.shape)
(16, 32, 7, 30, 30)
>>> # case 2: specify kernel_size with int, all parameters use default values.
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3)
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 30, 30, 30)
 >>> # case 3: stride=(1, 2, 3), other parameters being default.
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3, stride=(1, 2, 3))
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 30, 15, 10)
 >>> # case 4: pad_mode="pad", other parameters being default.
```

(continues on next page)

(continued from previous page)

```
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3, pad_mode="pad", pad=2)
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 34, 34, 34)
 >>> # case 5: dilation=(1, 1, 1), other parameters being default.
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3, dilation=(1, 1, 1))
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 30, 30, 30)
>>> # case 6: group=1, other parameters being default.
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3, group=1)
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 30, 30, 30)
>>> # case 7: All parameters are specified.
>>> x = Tensor(np.ones([10, 20, 32, 32, 32]), mindspore.float32)
>>> weight = Tensor(np.ones([40, 20, 3, 3, 3]), mindspore.float32)
>>> conv3d = ops.Conv3D(out_channel=40, kernel_size=3, stride=(1, 2, 3), pad_mode="pad",
...                     pad=2, dilation=(1), group=1)
>>> output = conv3d(x, weight)
>>> print(output.shape)
(10, 40, 34, 17, 12)
```

mindspore.ops.Conv3DTranspose

class mindspore.ops.**Conv3DTranspose**(*in_channel*, *out_channel*, *kernel_size*, *mode=1*, *pad_mode='valid'*, *pad=0*, *stride=1*, *dilation=1*, *group=1*, *output_padding=0*, *data_format='NCDHW'*)

Computes a 3D transposed convolution, which is also known as a deconvolution (although it is not an actual deconvolution).

Input is typically of shape (N, C, D, H, W), where N is batch size, C is channel number, D is depth, H is height, W is width.

If the 'pad_mode' is set to be "pad", the depth, height and width of output are defined as:

$$D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{pad}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$
$$H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{pad}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$
$$W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{pad}[2] + \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) + \text{output\_padding}[2] + 1$$

Note:

- In Ascend, only support $group = 1$.
- For Atlas A2 training series products, *output_padding* is currently not supported.

Parameters

- **in_channel** (*int*) −The channel of the input x.
- **out_channel** (*int*) −The channel of the weight x.

- **kernel_size** (*Union[int, tuple[int]]*) –The data type is int or a tuple of 3 integers. Specifies the depth, height and width of the 3D convolution window. Single int means the value is for the depth, height and width of the kernel. A tuple of 3 ints means the first value is for the depth, the second value is for the height and the other is for the width of the kernel.
- **mode** (*int, optional*) –Modes for different convolutions. Default is `1` . It is currently not used.
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"same"` , `"valid"` or `"pad"` . Default: `"valid"` .
  - `"same"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *pad* must be 0.
  - `"valid"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad* must be 0.
  - `"pad"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *pad* parameter. If this mode is set, *pad* must be greater than or equal to 0.
- **pad** (*Union(int, tuple[int]), optional*) –The pad value to be filled. Default: `0` . If *pad* is an integer, the paddings of head, tail, top, bottom, left and right are the same, equal to pad. If *pad* is a tuple of six integers, the padding of head, tail, top, bottom, left and right equal to pad[0], pad[1], pad[2], pad[3], pad[4] and pad[5] correspondingly.
- **stride** (*Union(int, tuple[int]), optional*) –The distance of kernel moving, an int number that represents the depth, height and width of movement are both strides, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: `1` .
- **dilation** (*Union(int, tuple[int]), optional*) –Specifies the space to use between kernel elements. Default: `1` .
- **group** (*int, optional*) –The number of groups into which the filter is divided. *in_channels* and *out_channels* must be divisible by *group*. Default: `1` .
- **output_padding** (*Union(int, tuple[int]), optional*) –Add extra size to each dimension of the output. Default: `0` .
- **data_format** (*str, optional*) –The optional value for data format. Currently only `'NCDHW'` is supported. Default: `'NCDHW'`.

Inputs:

- **dout** (Tensor) - The gradients with respect to the output of the convolution. The shape conforms to the default. data_format $(N, C_{in}, D_{out}, H_{out}, W_{out})$. Supported dtypes:
  - Ascend: float16.
  - GPU/CPU: float16, float32.
- **weight** (Tensor) - Set size of kernel is (K_d, K_h, K_w), then the shape is $(C_{in}, C_{out}//group, K_d, K_h, K_w)$. Where $group$ is the Args parameter, // is the symbol for integer division. It has the same dtype as *dout*.
- **bias** (Tensor) - Tensor of shape C_{out}. Currently, only support none. Default: `None` .

Outputs:

Tensor, the gradients with respect to the input of convolution 3D. Tensor of shape $(N, C_{out}//group, D_{out}, H_{out}, W_{out})$, where $group$ is the Args parameter.

Raises

- **TypeError** –If *in_channel*, *out_channel* or *group* is not an int.

- **TypeError** – If *kernel_size*, *stride*, *pad* , *dilation* or *output_padding* is neither an int not a tuple.
- **ValueError** – If *in_channel*, *out_channel*, *kernel_size*, *stride* or *dilation* is less than 1.
- **ValueError** – If *pad* is less than 0.
- **ValueError** – If *pad_mode* is not one of 'same', 'valid' nor 'pad'.
- **ValueError** – If *pad* is a tuple whose length is not equal to 6.
- **ValueError** – If *pad_mode* is not equal to 'pad' and *pad* is not equal to (0, 0, 0, 0, 0, 0).
- **ValueError** – If *data_format* is not 'NCDHW'.
- **TypeError** – If data type of dout and weight is neither float16 nor float32.
- **ValueError** – If bias is not none. The rank of dout and weight is not 5.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> dout = Tensor(np.ones([32, 16, 10, 32, 32]), mindspore.float16)
>>> weight = Tensor(np.ones([16, 3, 4, 6, 2]), mindspore.float16)
>>> conv3d_transpose = ops.Conv3DTranspose(in_channel=16, out_channel=3, kernel_size=(4, 6,␣
↪2))
>>> output = conv3d_transpose(dout, weight)
>>> print(output.shape)
(32, 3, 13, 37, 33)
```

mindspore.ops.CTCGreedyDecoder

class mindspore.ops.**CTCGreedyDecoder**(*merge_repeated=True*)

Performs greedy decoding on the logits given in inputs.

Refer to *mindspore.ops.ctc_greedy_decoder()* for more details.

Note: On Ascend, 'merge_repeated' can not be set to false.

Parameters

merge_repeated (*bool, optional*) – If `True` , merge repeated classes in output. Default: `True` .

Inputs:

- **inputs** (Tensor) - The input Tensor must be a 3-D tensor whose shape is $(max\_time, batch\_size, num\_classes)$. *num_classes* must be *num_labels* + *1* classes, *num_labels* indicates the number of actual labels. Blank labels are reserved. Default blank label is *num_classes* - *1*. Data type must be float32 or float64.
- **sequence_length** (Tensor) - A tensor containing sequence lengths with the shape of $(batch\_size,)$. The type must be int32. Each value in the tensor must be equal to or less than *max_time*.

Outputs:

- **decoded_indices** (Tensor) - A tensor with shape of $(total\_decoded\_outputs, 2)$. Data type is int64.
- **decoded_values** (Tensor) - A tensor with shape of $(total\_decoded\_outputs,)$, it stores the decoded classes. Data type is int64.
- **decoded_shape** (Tensor) - A tensor with shape of $(batch\_size, max\_decoded\_length)$. Data type is int64.
- **log_probability** (Tensor) - A tensor with shape of $(batch\_size, 1)$, containing sequence log-probability, has the same type as *inputs*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inputs = Tensor(np.array([[[0.6, 0.4, 0.2], [0.8, 0.6, 0.3]],
...                           [[0.0, 0.6, 0.0], [0.5, 0.4, 0.5]]]), mindspore.float32)
>>> sequence_length = Tensor(np.array([2, 2]), mindspore.int32)
>>> decoded_indices, decoded_values, decoded_shape, log_probability = ops.
 ↪CTCGreedyDecoder()(inputs,
...
 ↪sequence_length)
>>> print(decoded_indices)
[[0 0]
 [0 1]
 [1 0]]
>>> print(decoded_values)
[0 1 0]
>>> print(decoded_shape)
[2 2]
>>> print(log_probability)
[[-1.2]
 [-1.3]]
```

mindspore.ops.Dense

class mindspore.ops.**Dense**

```
prim = ops.Dense()
out = prim(input, weight, bias)
```

is equivalent to

```
ops.dense(input, weight, bias)
```

Refer to *mindspore.ops.dense()* for more details.

mindspore.ops.Dropout

class mindspore.ops.**Dropout**(*keep_prob=0.5*, *Seed0=0*, *Seed1=0*)

During training, randomly zeroes some of the elements of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution. It plays the role of reducing neuron correlation and avoid overfitting.

Refer to *mindspore.ops.dropout()* for more details.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *Seed0* and *Seed1* parameter have no effect.

Parameters

- **keep_prob** (*float, optional*) – The keep rate, between 0 and 1, e.g. keep_prob = 0.9, means dropping out 10% of input units. Default: `0.5`.
- **Seed0** (*int, optional*) – Seed0 value for random generating. Default: `0`.
- **Seed1** (*int, optional*) – Seed1 value for random generating. Default: `0`.

Inputs:

- **x** (Tensor) - The input Tensor of shape $(*, N)$, with data type of float16, float32 or float64.

Outputs:

- **output** (Tensor) - With the same shape and data type as *x*.
- **mask** (Tensor) - The mask applied to *x*.
  - On GPU and CPU, *mask* has the same shape and data type as *x*.
  - On Ascend, to achieve a better performance, it is denoted as a 1-D Tensor with Uint8 data type. It has shape $(byte\_counts,)$ where $byte\_counts$ is the number of bytes needed to mask the input *x*, $byte\_counts$ is calculated using the following formula:

    $$byte\_counts = \text{ceil}(\text{cumprod}(x.shape)/128) * 16$$

    If shape of *x* is $(2, 3, 4, 5, 6)$, the shape of *mask* will be $(96,)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> dropout = ops.Dropout(keep_prob=0.5)
>>> x = Tensor(np.ones([1, 2, 3, 4, 5]), mindspore.float32)
>>> output, mask = dropout(x)
>>> print(output.shape, mask.shape, mask.dtype)
(1, 2, 3, 4, 5) (16,) UInt8
```

mindspore.ops.Dropout2D

class mindspore.ops.**Dropout2D**(*keep_prob=0.5*)

During training, randomly zeroes some channels of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution(For a 4-dimensional tensor with a shape of (N, C, H, W), the channel feature map refers to a 2-dimensional feature map with the shape of (H, W)).

Dropout2D can improve the independence between channel feature maps.

Note: The keep probability $keep\_prob$ is equal to $1 - p$ in *mindspore.ops.dropout2d()*.

Parameters

keep_prob (*float, optional*) – The keep probability of a channel, between 0 and 1, e.g. *keep_prob* = 0.8, means dropping out 20% of channels. Default: `0.5` .

Inputs:

- **x** (Tensor) - A 4-D tensor with shape (N, C, H, W), where N is the batch size, C is the number of channels, H is the feature height, and W is the feature width.

Outputs:

- **output** (Tensor) - With the same shape and data type as *x*.
- **mask** (Tensor) - With the same shape as *x* and the data type is bool.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If the data type of *keep_prob* is not float.
- **ValueError** – If *keep_prob* is out of the range *[0.0, 1.0]*.
- **ValueError** – If *x* shape is not *4D*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> dropout = ops.Dropout2D(keep_prob=0.5)
>>> x = Tensor(np.ones([2, 1, 2, 3]), mindspore.float32)
>>> output, mask = dropout(x)
>>> print(output.shape)
(2, 1, 2, 3)
```

mindspore.ops.Dropout3D

class mindspore.ops.**Dropout3D**(*keep_prob=0.5*)

During training, randomly zeroes some channels of the input tensor with probability $1 - keep\_prob$ from a Bernoulli distribution(For a 5-dimensional tensor with a shape of NCDHW, the channel feature map refers to a 3-dimensional feature map with a shape of DHW).

Note: The keep probability $keep\_prob$ is equal to $1 - p$ in *mindspore.ops.dropout3d()*.

Dropout3D can improve the independence between channel feature maps.

Parameters

keep_prob (*float, optional*) – The keep probability of a channel, between 0 and 1, e.g. *keep_prob* = 0.8, means dropping out 20% of channels. Default: `0.5` .

Inputs:

- **x** (Tensor) - A 5-D tensor with shape (N, C, D, H, W), where N is the batch size, C is the number of channels, D is the feature depth, H is the feature height, and W is the feature width.

Outputs:

- **output** (Tensor) - With the same shape and data type as *x*.
- **mask** (Tensor) - With the same shape as *x* and the data type is bool.

Raises

- **TypeError** – If the data type of *keep_prob* is not float.
- **ValueError** – If *keep_prob* is out of the range [0.0, 1.0]; or if the dim of input is not 5-D.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> dropout = ops.Dropout3D(keep_prob=0.5)
>>> x = Tensor(np.ones([2, 1, 2, 1, 2]), mindspore.float32)
>>> output, mask = dropout(x)
>>> print(output.shape)
(2, 1, 2, 1, 2)
```

mindspore.ops.DynamicGRUV2

class mindspore.ops.**DynamicGRUV2**(*direction='UNIDIRECTIONAL'*, *cell_depth=1*, *keep_prob=1.0*, *cell_clip=- 1.0*, *num_proj=0*, *time_major=True*, *activation='tanh'*, *gate_order='rzh'*, *reset_after=True*, *is_training=True*)

Applies a single-layer gated recurrent unit (GRU) to an input sequence.

$$
\begin{aligned}
r_{t+1} &= \sigma(W_{ir}x_{t+1} + b_{ir} + W_{hr}h_{(t)} + b_{hr}) \\
z_{t+1} &= \sigma(W_{iz}x_{t+1} + b_{iz} + W_{hz}h_{(t)} + b_{hz}) \\
n_{t+1} &= \tanh(W_{in}x_{t+1} + b_{in} + r_{t+1} * (W_{hn}h_{(t)} + b_{hn})) \\
h_{t+1} &= (1 - z_{t+1}) * n_{t+1} + z_{t+1} * h_{(t)}
\end{aligned}
$$

where h_{t+1} is the hidden state at time *t+1*, x_{t+1} is the input at time *t+1*, h_t is the hidden state of the layer at time *t* or the initial hidden state at time *0*. r_{t+1}, z_{t+1}, n_{t+1} are the reset, update, and new gates, respectively. W, b are the weight parameter and the deviation parameter respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

Parameters

- **direction** (*str, optional*) – A string identifying the direction in the operator. Default: `'UNIDIRECTIONAL'`. Only `'UNIDIRECTIONAL'` is currently supported.
- **cell_depth** (*int, optional*) – An integer identifying the cell depth in the operator. Default: `1`.
- **keep_prob** (*float, optional*) – A float identifying the keep prob in the operator. Default: `1.0`.
- **cell_clip** (*float, optional*) – A float identifying the cell clip in the operator. Default: `-1.0`.
- **num_proj** (*int, optional*) – An integer identifying the number projection in the operator. Default: `0`.
- **time_major** (*bool, optional*) – A bool identifying the time major in the operator. Default: `True`.
- **activation** (*str, optional*) – A string identifying the type of activation function in the operator. Default: `'tanh'`. Only `'tanh'` is currently supported.
- **gate_order** (*str, optional*) – A string identifying the gate order in weight and bias. Default: `'rzh'`. `'zrh'` is another option. Here, `'rzh'` means the gate order is: reset gate, update gate, hidden gate. `'zrh'` means the gate order is: update gate, reset gate, hidden gate.
- **reset_after** (*bool, optional*) – A bool identifying whether to apply reset gate after matrix multiplication. Default: `True`.
- **is_training** (*bool, optional*) – A bool identifying is training in the operator. Default: `True`.

Inputs:

- **x** (Tensor) - Current words. Tensor of shape (num_step, batch_size, input_size). The data type must be float16.
- **weight_input** (Tensor) - Input-hidden weight $W_{\{ir,iz,in\}}$. Tensor of shape (input_size, $3 \times$ hidden_size). The data type must be float16.
- **weight_hidden** (Tensor) - Hidden-hidden weight $W_{\{hr,hz,hn\}}$. Tensor of shape (hidden_size, $3 \times$ hidden_size). The data type must be float16.
- **bias_input** (Tensor) - Input-hidden bias $b_{\{ir,iz,in\}}$. Tensor of shape ($3 \times$ hidden_size), or None. Has the same data type with input *init_h*.
- **bias_hidden** (Tensor) - Hidden-hidden bias $b_{\{hr,hz,hn\}}$. Tensor of shape ($3 \times$ hidden_size), or None. Has the same data type with input *init_h*.
- **seq_length** (Tensor) - The length of each batch. Tensor of shape (batch_size). Only *None* is currently supported.
- **init_h** (Tensor) - Hidden state of initial time. Tensor of shape (batch_size, hidden_size). The data type must be float16 or float32.

Outputs:

- **y** (Tensor) - A Tensor of shape:
  - y_shape = $(num\_step, batch\_size, min(hidden\_size, num\_proj))$: *If num_proj > 0*,
  - y_shape = $(num\_step, batch\_size, hidden\_size)$: *If num_proj = 0*.

  Has the same data type with input *bias_type*.
- **output_h** (Tensor) - A Tensor of shape (num_step, batch_size, hidden_size). Has the same data type with input *bias_type*.
- **update** (Tensor) - A Tensor of shape (num_step, batch_size, hidden_size). Has the same data type with input *bias_type*.
- **reset** (Tensor) - A Tensor of shape (num_step, batch_size, hidden_size). Has the same data type with input *bias_type*.
- **new** (Tensor) - A Tensor of shape (num_step, batch_size, hidden_size). Has the same data type with input *bias_type*.
- **hidden_new** (Tensor) - A Tensor of shape (num_step, batch_size, hidden_size). Has the same data type with input *bias_type*.

A note about the bias_type:

- If *bias_input* and *bias_hidden* both are *None*, *bias_type* is the data type of *init_h*.
- If *bias_input* is not *None*, *bias_type* is the data type of *bias_input*.
- If *bias_input* is *None* and *bias_hidden* is not *None*, *bias_type* is the data type of *bias_hidden*.

Raises

- **TypeError** –If *direction*, *activation* or *gate_order* is not a str.
- **TypeError** –If *cell_depth* or *num_proj* is not an int.
- **TypeError** –If *keep_prob* or *cell_clip* is not a float.
- **TypeError** –If *time_major*, *reset_after* or *is_training* is not a bool.
- **TypeError** –If *x*, *weight_input*, *weight_hidden*, *bias_input*, *bias_hidden*, *seq_length* or *ini_h* is not a Tensor.
- **TypeError** –If dtype of *x*, *weight_input* or *weight_hidden* is not float16.
- **TypeError** –If dtype of *init_h* is neither float16 nor float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.rand(2, 8, 64).astype(np.float16))
>>> weight_i = Tensor(np.random.rand(64, 48).astype(np.float16))
>>> weight_h = Tensor(np.random.rand(16, 48).astype(np.float16))
>>> bias_i = Tensor(np.random.rand(48).astype(np.float16))
>>> bias_h = Tensor(np.random.rand(48).astype(np.float16))
>>> init_h = Tensor(np.random.rand(8, 16).astype(np.float16))
>>> dynamic_gru_v2 = ops.DynamicGRUV2()
>>> output = dynamic_gru_v2(x, weight_i, weight_h, bias_i, bias_h, None, init_h)
>>> print(output[0].shape)
(2, 8, 16)
```

mindspore.ops.DynamicRNN

class mindspore.ops.**DynamicRNN**(*cell_type='LSTM'*, *direction='UNIDIRECTIONAL'*, *cell_depth=1*, *use_peephole=False*, *keep_prob=1.0*, *cell_clip=- 1.0*, *num_proj=0*, *time_major=True*, *activation='tanh'*, *forget_bias=0.0*, *is_training=True*)

Applies a recurrent neural network to the input. Only long short-term memory (LSTM) is supported currently.

$$
\begin{aligned}
i_{t+1} &= \sigma(W_{ix}x_{t+1} + b_{ix} + W_{ih}h_{(t)} + b_{ih}) \\
f_{t+1} &= \sigma(W_{fx}x_{t+1} + b_{fx} + W_{fh}h_{(t)} + b_{fh}) \\
\tilde{c}_{t+1} &= \tanh(W_{cx}x_{t+1} + b_{cx} + W_{ch}h_{(t)} + b_{ch}) \\
o_{t+1} &= \sigma(W_{ox}x_{t+1} + b_{ox} + W_{oh}h_{(t)} + b_{oh}) \\
c_{t+1} &= f_{t+1} * c_{(t)} + i_t * \tilde{c}_{t+1} \\
h_{t+1} &= o_{t+1} * \tanh(c_{t+1})
\end{aligned}
$$

h_{t+1} is the hidden state at time *t+1*. x_{t+1} is the input at time *t+1*. h_t is the hidden state of the layer at time *t* or the initial hidden state at time *0*. σ is the sigmoid function, and $*$ is the Hadamard product. W, b are learnable weights between the output and the input in the formula. For instance, W_{ix}, b_{ix} are the weight and bias used to transform from input x to i.

Parameters

- **cell_type** (*str, optional*) – A string identifying the cell type in the operator. Default: `'LSTM'`. Only 'LSTM' is currently supported.
- **direction** (*str, optional*) – A string identifying the direction in the operator. Default: `'UNIDIRECTIONAL'`. Only 'UNIDIRECTIONAL' is currently supported.
- **cell_depth** (*int, optional*) – An integer identifying the cell depth in the operator. Default: `1`.
- **use_peephole** (*bool, optional*) – A bool identifying if use peephole in the operator. Default: `False`.
- **keep_prob** (*float, optional*) – A float identifying the keep prob in the operator. Default: `1.0`.
- **cell_clip** (*float, optional*) – A float identifying the cell clip in the operator. Default: `-1.0`.
- **num_proj** (*int, optional*) – An integer identifying the number projection in the operator. Default: `0`.
- **time_major** (*bool, optional*) – A bool specify the data format of *x*. If it is set to `True`, the format is ($num\_step, batch\_size, input\_size$), if it is set to False, the format is ($batch\_size, num\_step, input\_size$). Default: `True`. Only supports `True` at present.
- **activation** (*str, optional*) – A string identifying the type of activation function in the operator. Default: `'tanh'`. Only 'tanh' is currently supported.
- **forget_bias** (*float, optional*) – A float identifying the forget bias in the operator. Default: `0.0`.
- **is_training** (*bool, optional*) – A bool identifying is training in the operator. Default: `True`.

Inputs:

- **x** (Tensor) - Current words. Tensor of shape ($num\_step, batch\_size, input\_size$). The data type must be float16.
- **w** (Tensor) - Weight. Tensor of shape ($input\_size + hidden\_size, 4 * hidden\_size$). The data type must be float16.
- **b** (Tensor) - Bias. Tensor of shape ($4 * hidden\_size$). The data type must be float16.
- **seq_length** (Tensor) - The length of each batch. Tensor of shape ($batch\_size,$). Only *None* is currently supported.
- **init_h** (Tensor) - Hidden state of initial time. Tensor of shape ($1, batch\_size, hidden\_size$). The data type must be float16.
- **init_c** (Tensor) - Cell state of initial time. Tensor of shape ($1, batch\_size, hidden\_size$). The data type must be float16.

Outputs:

- **y** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **output_h** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). With data type of float16.
- **output_c** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **i** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **j** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **f** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **o** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.
- **tanhct** (Tensor) - A Tensor of shape ($num\_step, batch\_size, hidden\_size$). Has the same type with input *b*.

Raises

- **TypeError** – If *cell_type*, *direction* or *activation* is not a str.
- **TypeError** – If *cell_depth* or *num_proj* is not an int.
- **TypeError** – If *keep_prob*, *cell_clip* or *forget_bias* is not a float.
- **TypeError** – If *use_peehpole*, *time_major* or *is_training* is not a bool.
- **TypeError** – If *x*, *w*, *b*, *seq_length*, *init_h* or *init_c* is not a Tensor.
- **TypeError** – If dtype of *x*, *w*, *init_h* or *init_c* is not float16.
- **TypeError** – If dtype of *b* is neither float16 nor float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.rand(2, 16, 64).astype(np.float16))
>>> w = Tensor(np.random.rand(96, 128).astype(np.float16))
>>> b = Tensor(np.random.rand(128).astype(np.float16))
>>> init_h = Tensor(np.random.rand(1, 16, 32).astype(np.float16))
>>> init_c = Tensor(np.random.rand(1, 16, 32).astype(np.float16))
>>> dynamic_rnn = ops.DynamicRNN()
>>> output = dynamic_rnn(x, w, b, None, init_h, init_c)
>>> print(output[0].shape)
(2, 16, 32)
```

mindspore.ops.Flatten

class mindspore.ops.**Flatten**

Flattens a tensor without changing its batch size on the 0-th axis.

Refer to *mindspore.ops.flatten()* for more details.

Inputs:

- **input_x** (Tensor) - Tensor of shape $(N, \ldots)$ to be flattened, where N is batch size.

Outputs:

Tensor, the shape of the output tensor is (N, X), where X is the product of the remaining dimension.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones(shape=[1, 2, 3, 4]), mindspore.float32)
>>> flatten = ops.Flatten()
>>> output = flatten(input_x)
>>> print(output.shape)
(1, 24)
```

mindspore.ops.FractionalMaxPool3DWithFixedKsize

class mindspore.ops.**FractionalMaxPool3DWithFixedKsize**(*ksize*, *output_shape*, *data_format='NCDHW'*)

Applies a 3D fractional max pooling to an input signal composed of multiple input planes. The max-pooling operation is applied in (kD, kH, kW) regions by a stochastic step size determined by the target output size *output_shape*.

The number of output features is equal to the number of input planes.

Refer to the paper Fractional MaxPooling by Ben Graham for more details.

The input and output data format can be "NCDHW" and "NDHWC". N is the batch size, C is the number of channels, D the feature depth, H is the feature height, and W is the feature width.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **ksize** (*Union[float, tuple]*) –Size of the pooling window. *ksize* can be a tuple of three values specify a shape (k_D, k_H, k_W), or a single int K for (K, K, K).
- **output_shape** (*Union[int, tuple]*) –The target output shape. *output_shape* can be a tuple of three values specify a shape $(D_{out}, H_{out}, W_{out})$, or a single float S for (S, S, S).
- **data_format** (*str, optional*) –The optional value for data format. Currently support `'NCDHW'` and `'NHDWC'` . Default: `'NCDHW'` .

Inputs:

- **x** (Tensor) - The input of FractionalMaxPool3DWithFixedKsize, which is a 4D or 5D tensor. Tensor of data type : float16, float32, double, int32, int64. Supported shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(N, D_{in}, H_{in}, W_{in}, C)$.
- **random_samples** (Tensor) - The random step of FractionalMaxPool3DWithFixedKsize, which is a 3D tensor. Tensor of data type : float16, float32, double, and value is between (0, 1). Supported shape $(N, C, 3)$

Outputs:

- **y** (Tensor) - A tensor, the output of FractionalMaxPool3DWithFixedKsize. Has the same data type with *x*. Tensor of shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(N, D_{out}, H_{out}, W_{out}, C)$.
- **argmax** (Tensor) - A tensor, the indices along with the outputs. Has the same shape as the *y* and int32 or int64 data type.

Raises

- **TypeError** –If *input_x* is not a 4D or 5D tensor.
- **TypeError** –If *random_samples* is not a 3D tensor.
- **TypeError** –If data type of *x* is not float16, float32, double, int32, int64.
- **TypeError** –If dtype of *random_samples* is not float16, float32, double.
- **TypeError** –If dtype of *argmax* is not int32, int64.
- **ValueError** –If *output_shape* is a tuple and if *output_shape* length is not 3.
- **ValueError** –If *ksize* is a tuple and if *ksize* length is not 3.
- **ValueError** –If numbers in *output_shape* or *ksize* is not positive.
- **ValueError** –If *data_format* is neither 'NCDHW' nor 'NDHWC'.
- **ValueError** –If the first dimension size of *input_x* and *random_samples* is not equal.
- **ValueError** –If the second dimension size of *input_x* and *random_samples* is not equal.
- **ValueError** –If the third dimension size of *random_samples* is not 3.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
...       .reshape([1, 1, 2, 2, 4]), mstype.float32)
>>> random_samples = Tensor(np.array([0.7, 0.7, 0.7]).reshape([1, 1, 3]), mstype.float32)
>>> ksize = (1, 1, 1)
>>> output_shape = (1, 1, 2)
>>> net = ops.FractionalMaxPool3DWithFixedKsize(ksize = ksize, output_shape = output_shape)
>>> output, argmax = net(x, random_samples)
>>> print(output)
[[[[[13. 16.]]]]]
>>> print(argmax)
[[[[[12 15]]]]]
```

mindspore.ops.GridSampler2D

class mindspore.ops.**GridSampler2D**(*interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*)

This operation samples 2d *input_x* by using interpolation based on flow field grid, which is usually gennerated by *mindspore.ops.affine_grid()*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.grid_sample()* for more details.

Parameters

- **interpolation_mode** (*str, optional*) – An optional string specifying the interpolation method. The optional values are "bilinear" or "nearest" . Default: "bilinear" .
  - "nearest": Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.
  - "bilinear": Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels, computed using bilinear interpolation. This method produces smoother results compared to nearest neighbor interpolation.
- **padding_mode** (*str, optional*) – An optional string specifying the pad method. The optional values are "zeros" , "border" or "reflection" . Default: "zeros" . When the sampling grid is outside input's bounds, effects of various padding modes are as follows:
  - "zeros": Pads the input tensor with zeros.
  - "border": Pads the input tensor with the values of the pixels on the border of the tensor.
  - "reflection": Pads the input tensor by reflecting the values of the pixels at the boundary of the tensor.
- **align_corners** (*bool, optional*) – An optional bool. When set to True , the centers of the corner pixels of the input and output tensors are aligned. When set to False , it is not aligned. Default: False .

Inputs:

- **input_x** (Tensor) - A 4-D tensor with shape (N, C, H_{in}, W_{in}). Supported dtypes:
  - Ascend: float16, float32.
  - GPU/CPU: float16, float32, float64.
- **grid** (Tensor) - A 4-D tensor whose dtype is the same as *input_x* and whose shape is $(N, H_{out}, W_{out}, 2)$. Used to specify the sampling pixel locations normalized by the input spatial dimensions.

Outputs:

A 4-D Tensor whose dtype is the same as *input_x* and whose shape is (N, C, H_{out}, W_{out}).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> gridsampler = ops.GridSampler2D(interpolation_mode='bilinear', padding_mode='zeros',␣
↪align_corners=True)
>>> input_x = Tensor(np.arange(16).reshape((2, 2, 2, 2)).astype(np.float32))
>>> grid = Tensor(np.arange(-9, 9, 0.5).reshape((2, 3, 3, 2)).astype(np.float32))
>>> output = gridsampler(input_x, grid)
>>> print(output)
[[[[ 0.     0.     0.   ]
   [ 0.     0.     0.   ]
   [ 0.     0.     0.5  ]]
  [[ 0.     0.     0.   ]
   [ 0.     0.     0.   ]
   [ 0.     1.5    4.5  ]]]
 [[[10.     8.25   1.375]
   [ 0.     0.     0.   ]
   [ 0.     0.     0.   ]]
  [[14.    11.25   1.875]
   [ 0.     0.     0.   ]
   [ 0.     0.     0.   ]]]]
```

mindspore.ops.GridSampler3D

class mindspore.ops.**GridSampler3D**(*interpolation_mode='bilinear'*, *padding_mode='zeros'*, *align_corners=False*)

Given an input and a grid, the output is calculated using the input values and pixel positions in the grid. Only volume (5-D) input is supported.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.grid_sample()* for more details.

Parameters

- **interpolation_mode** (*str, optional*) – An optional string specifying the interpolation method. The optional values are `"bilinear"` or `"nearest"` . Default: `"bilinear"` .
  - `"nearest"`: Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.
  - `"bilinear"`: Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels. This method produces smoother results compared to nearest neighbor interpolation.
- **padding_mode** (*str, optional*) – An optional string specifying the pad method. The optional values are `"zeros"` , `"border"` or `"reflection"` . Default: `"zeros"` . When the sampling grid is outside input's bounds, effects of various padding modes are as follows:
  - `"zeros"`: Pads the input tensor with zeros.
  - `"border"`: Pads the input tensor with the values of the pixels on the border of the tensor.
  - `"reflection"`: Pads the input tensor by reflecting the values of the pixels at the boundary of the tensor.
- **align_corners** (*bool, optional*) – An optional bool specifying alignment method. If set to `True` , the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to `False` , they are

instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. Default: `False` .

Inputs:

- **input_x** (Tensor) - A 5-D tensor with dtype of float16, float32 or float64 and shape of $(N, C, D_{in}, H_{in}, W_{in})$.
- **grid** (Tensor) - A 5-D tensor whose dtype is the same as *input_x* and whose shape is $(N, D_{out}, H_{out}, W_{out}, 3)$.

Outputs:

A 5-D Tensor whose dtype is the same as *input_x* and whose shape is $(N, C, D_{out}, H_{out}, W_{out})$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> gridsampler = ops.GridSampler3D(interpolation_mode='bilinear', padding_mode='zeros',␣
↪align_corners=True)
>>> input_x = Tensor(np.arange(32).reshape((2, 2, 2, 2, 2)).astype(np.float32))
>>> grid = Tensor(np.arange(-0.2, 1, 0.1).reshape((2, 2, 1, 1, 3)).astype(np.float32))
>>> output = gridsampler(input_x, grid)
>>> print(output)
[[[[[ 3.3     ]]
   [[ 4.35    ]]]
  [[[11.300001]]
   [[12.349999]]]]
 [[[[21.4     ]]
   [[22.449999]]]
  [[[29.4     ]]
   [[30.449999]]]]]
```

mindspore.ops.LayerNorm

class mindspore.ops.**LayerNorm**(*begin_norm_axis=1*, *begin_params_axis=1*, *epsilon=1e-7*)

Applies the Layer Normalization to the input tensor.

This operator will normalize the input tensor on given axis. LayerNorm is described in the paper Layer Normalization.

$$y = \frac{x - mean}{\sqrt{variance + \epsilon}} * \gamma + \beta$$

where γ is scale, β is bias, ϵ is epsilon.

Parameters

- **begin_norm_axis** (*int*) –The begin axis of the *input_x* to apply LayerNorm, the value must be in [-1, rank(input_x)). Default: `1` .
- **begin_params_axis** (*int*) –The begin axis of the parameter input (*gamma*, *beta*) to apply LayerNorm, the value must be in [-1, rank(input_x)). Default: `1` . Note: On the Ascend platform, the value of *begin_params_axis* needs to be equal to the value of *begin_norm_axis* .
- **epsilon** (*float*) –A value added to the denominator for numerical stability(ϵ). Default: `1e-7` .

Inputs:

- **input_x** (Tensor) - Tensor of shape $(N, \ldots)$. The input of LayerNorm. Supported dtypes: float16, float32, float64.
- **gamma** (Tensor) - Learnable parameter γ . Tensor of shape *input_x_shape[begin_params_axis:]*. Supported dtypes: float16, float32, float64.
- **beta** (Tensor) - Learnable parameter β . Tensor of shape *input_x_shape[begin_params_axis:]*. Supported dtypes: float16, float32, float64.

Outputs:

tuple[Tensor], tuple of 3 tensors, the normalized input and the updated parameters.

- **output_x** (Tensor) - The normalized input, has the same type and shape as the *input_x*.
- **mean** (Tensor) - The first *begin_norm_axis* dimensions of *mean* shape is the same as *input_x*, and the remaining dimensions are 1. Suppose the shape of the *input_x* is $(x_1, x_2, \ldots, x_R)$, the shape of the *mean* is $(x_1, \ldots, x_{begin\_norm\_axis}, 1, \ldots, 1)$ (when *begin_norm_axis=0*, the shape of *mean* is $(1, \ldots, 1)$).
- **rstd** (Tensor) - The reciprocal of the input standard deviation. Shape is the same as *mean* .

Raises

- **TypeError** –If *begin_norm_axis* or *begin_params_axis* is not an int.
- **TypeError** –If *epsilon* is not a float.
- **TypeError** –If *input_x*, *gamma* or *beta* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[1, 2, 3], [1, 2, 3]]), mindspore.float32)
>>> gamma = Tensor(np.ones([3]), mindspore.float32)
>>> beta = Tensor(np.ones([3]), mindspore.float32)
>>> layer_norm = ops.LayerNorm()
>>> output, _, _ = layer_norm(input_x, gamma, beta)
>>> print(output)
[[-0.2247448  1.         2.2247448]
 [-0.2247448  1.         2.2247448]]
```

mindspore.ops.LRN

class mindspore.ops.**LRN**(*depth_radius=5*, *bias=1.0*, *alpha=1.0*, *beta=0.5*, *norm_region='ACROSS_CHANNELS'*)

Local Response Normalization.

> **Warning:** LRN is deprecated on Ascend due to potential accuracy problem. It's recommended to use other normalization methods, e.g. *mindspore.ops.BatchNorm*.

$$b_c = a_c \left(k + \frac{\alpha}{n} \sum_{c'=\max(0,c-n/2)}^{\min(N-1,c+n/2)} a_{c'}^2 \right)^{-\beta}$$

where the a_c indicates the specific value of the pixel corresponding to c in feature map; where the $n/2$ indicates the *depth_radius*; where the k indicates the *bias*; where the α indicates the *alpha*; where the β indicates the *beta*.

Parameters

- **depth_radius** (*int*) –Half-width of the 1-D normalization window with the shape of 0-D. Default: `5` .
- **bias** (*float*) –An offset (usually positive to avoid dividing by 0). Default: `1.0` .
- **alpha** (*float*) –A scale factor, usually positive. Default: `1.0` .
- **beta** (*float*) –An exponent. Default: `0.5` .
- **norm_region** (*str*) –Specifies normalization region. Options: `"ACROSS_CHANNELS"` . Default: `"ACROSS_CHANNELS"` .

Inputs:

- **x** (Tensor) - A 4-D Tensor with float16 or float32 data type.

Outputs:

Tensor, with the same shape and data type as *x*.

Raises

- **TypeError** –If *depth_radius* is not an int.
- **TypeError** –If *bias*, *alpha* or *beta* is not a float.
- **TypeError** –If *norm_region* is not a str.
- **TypeError** –If *x* is not a Tensor.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[[0.1], [0.2]],
...                       [[0.3], [0.4]]]]), mindspore.float32)
>>> lrn = ops.LRN()
>>> output = lrn(x)
>>> print(output)
[[[[0.09534626]
   [0.1825742 ]]
  [[0.2860388 ]
   [0.3651484 ]]]]
```

mindspore.ops.LSTM

class mindspore.ops.**LSTM**(*input_size*, *hidden_size*, *num_layers*, *has_bias*, *bidirectional*, *dropout*, *proj_size=0*)

Performs the Long Short-Term Memory (LSTM) on the input.

For more information, please refer to *mindspore.nn.LSTM*.

Parameters

- **input_size** (*int*) −Number of features of input.
- **hidden_size** (*int*) −Number of features of hidden layer.
- **num_layers** (*int*) −Number of layers of stacked LSTM, , which is only support *1* on CPU.
- **has_bias** (*bool*) −Whether the cell has bias *b_ih* and *b_hh* , which is only support *False* on CPU.
- **bidirectional** (*bool*) −Specifies whether it is a bidirectional LSTM, , which is only support *False* on CPU.
- **dropout** (*float*) −If not 0, append *Dropout* layer on the outputs of each LSTM layer except the last layer. The range of dropout is [0.0, 1.0].
- **proj_size** (*int*) −If *proj_size* > 0, a projection of the corresponding size will be used, which is only supported on CPU now. Default: `0` .

Inputs:

- **input** (Tensor) - Tensor of shape $(seq\_len, batch\_size, input\_size)$ or $(batch\_size, seq\_len, input\_size)$.
- **h** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, real\_hidden\_size)$.
- **c** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.
- **w** (Tensor) - A weight Tensor.

If $proj\_size > 0$, $real\_hidden\_size = proj\_size$, otherwise $real\_hidden\_size = hidden\_size$.

Outputs:

Tuple, a tuple contains *(output, h_n, c_n, reserve, state)*.

- **output** (Tensor) - Tensor of shape $(seq\_len, batch\_size, num\_directions * real\_hidden\_size)$.
- **h_n** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, real\_hidden\_size)$.
- **c_n** (Tensor) - Tensor of shape $(num\_directions * num\_layers, batch\_size, hidden\_size)$.
- **reserve** (Tensor) - Tensor of shape $(r, 1)$.
- **state** (Tensor) - Random number generator state and its shape is $(s, 1)$.

Raises

- **TypeError** −If *input_size*, *hidden_size* or *num_layers* is not an int.
- **TypeError** −If *has_bias* or *bidirectional* is not a bool.
- **TypeError** −If *dropout* is not a float.
- **ValueError** −If *dropout* is not in range [0.0, 1.0].
- **ValueError** −If *proj_size* is not in range [0, *hidden_size*).

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_size = 10
>>> hidden_size = 2
>>> num_layers = 1
>>> seq_len = 5
>>> batch_size = 2
>>>
>>> net = ops.LSTM(input_size, hidden_size, num_layers, True, False, 0.0)
>>> input_tensor = Tensor(np.ones([seq_len, batch_size, input_size]).astype(np.float32))
>>> h0 = Tensor(np.ones([num_layers, batch_size, hidden_size]).astype(np.float32))
>>> c0 = Tensor(np.ones([num_layers, batch_size, hidden_size]).astype(np.float32))
>>> w = Tensor(np.ones([112, 1, 1]).astype(np.float32))
>>> output, hn, cn, _, _ = net(input_tensor, h0, c0, w)
>>> print(output)
[[[0.9640267  0.9640267 ]
  [0.9640267  0.9640267 ]]
 [[0.9950539  0.9950539 ]
  [0.9950539  0.9950539 ]]
 [[0.99932843 0.99932843]
  [0.99932843 0.99932843]]
 [[0.9999084  0.9999084 ]
  [0.9999084  0.9999084 ]]
 [[0.9999869  0.9999869 ]
  [0.9999869  0.9999869 ]]]
```

mindspore.ops.MaxPool

class mindspore.ops.**MaxPool**(*kernel_size=1*, *strides=1*, *pad_mode='valid'*, *data_format='NCHW'*)

Max pooling operation.

Applies a 2D max pooling over an input Tensor which can be regarded as a composition of 2D planes.

Typically the input is of shape $(N_{in}, C_{in}, H_{in}, W_{in})$, MaxPool outputs regional maximum in the (H_{in}, W_{in})-dimension. Given kernel size $ks = (h_{ker}, w_{ker})$ and stride $s = (s_0, s_1)$, the operation is as follows:

$$\text{output}(N_i, C_j, h, w) = \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times h + m, s_1 \times w + n)$$

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value, is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively. Default: `1` .
- **strides** (*Union[int, tuple[int]]*) – The distance of kernel moving, an int number that represents not only the height of movement but also the width of movement, or a tuple of two int numbers that represent height and width of movement respectively. Default: `1` .
- **pad_mode** (*str, optional*) – Specifies the padding mode with a padding value of 0. It can be set to: `'same'` or `'valid'` . Default: `'valid'` .
  - `'same'`: Pad the input around its edges so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally, If the amount is even, it is uniformly distributed around the input, if it is odd, the excess amount goes to the right/bottom side.
  - `'valid'`: No padding is applied to the input, and the output returns the maximum possible height and width. Extra pixels that could not complete a full stride will be discarded.

- **data_format** (*str*) –The optional value for data format, is `'NHWC'` or `'NCHW'` . Default: `'NCHW'` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C_{in}, H_{in}, W_{in})$. Supported dtypes:
  - CPU: float16, float32, float64.
  - GPU/Ascend: float16, float32.

Outputs:

Tensor, with shape $(N, C_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** –If *kernel_size* or *strides* is neither int nor tuple.
- **ValueError** –If *pad_mode* is neither `'valid'` nor `'same'` with not case sensitive.
- **ValueError** –If *data_format* is neither `'NCHW'` nor `'NHWC'`.
- **ValueError** –If *kernel_size* or *strides* is less than 1.
- **ValueError** –If length of shape of *input* is not equal to 4.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(1 * 3 * 3 * 4).reshape((1, 3, 3, 4)), mindspore.float32)
>>> maxpool_op = ops.MaxPool(pad_mode="VALID", kernel_size=2, strides=1)
>>> output = maxpool_op(x)
>>> print(output)
[[[[ 5.  6.  7.]
   [ 9. 10. 11.]]
  [[17. 18. 19.]
   [21. 22. 23.]]
  [[29. 30. 31.]
   [33. 34. 35.]]]]
```

mindspore.ops.MaxPool3D

class mindspore.ops.**MaxPool3D**(*kernel_size=1*, *strides=1*, *pad_mode='VALID'*, *pad_list=0*, *ceil_mode=None*, *data_format='NCDHW'*)

Applies a 3D max pooling over an input Tensor which can be regarded as a composition of 3D planes.

Typically the input is of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$, MaxPool outputs regional maximum in the (D_{in}, H_{in}, W_{in})-dimension. Given kernel size $ks = (d_{ker}, h_{ker}, w_{ker})$ and stride $s = (s_0, s_1, s_2)$, the operation is as follows:

$$\text{output}(N_i, C_j, d, h, w) = \max_{l=0,\ldots,d_{ker}-1} \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

Note: For Atlas training series products, this primitive is not supported.

Parameters

- **kernel_size** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value, is an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively. Default: `1` .
- **strides** (*Union[int, tuple[int]]*) –The distance of kernel moving, an int number that represents not only the depth, height of movement but also the width of movement,, or a tuple of three int numbers that represent depth, height and width of movement respectively. Default: `1` .
- **pad_mode** (*str, optional*) –Specifies the padding mode with a padding value of 0. It can be set to: `"SAME"` , `"VALID"` or `"PAD"` . Default: `"VALID"` .
  - `"SAME"`: Pad the input around its depth/height/width dimension so that the shape of input and output are the same when *stride* is set to `1`. The amount of padding to is calculated by the operator internally. If the amount is even, it isuniformly distributed around the input, if it is odd, the excess amount goes to the front/right/bottom side. If this mode is set, *pad_list* must be 0.
  - `"VALID"`: No padding is applied to the input, and the output returns the maximum possible depth, height and width. Extra pixels that could not complete a full stride will be discarded. If this mode is set, *pad_list* must be 0.
  - `"PAD"`: Pad the input with a specified amount. In this mode, the amount of padding in the depth, height and width dimension is determined by the *pad_list* parameter. If this mode is set, *pad_list* must be greater than or equal to 0.
- **pad_list** (*Union(int, tuple[int])*) –The pad value to be filled. Default: `0` . If *pad* is an integer, the paddings of head, tail, top, bottom, left and right are the same, equal to pad. If *pad* is a tuple of six integers, the padding of head, tail, top, bottom, left and right equals to pad[0], pad[1], pad[2], pad[3], pad[4] and pad[5] correspondingly.
- **ceil_mode** (*Union[bool, None]*) –Whether to use ceil instead of floor to calculate output shape. Only effective in "pad" mode. When *pad_mode* is `"pad"` and "ceil_mode" is `None` , *ceil_mode* will be set as `False`. Default: `None` .
- **data_format** (*str*) –The optional value for data format. Currently only support `"NCDHW"` . Default: `"NCDHW"` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$. Data type must be float16, float32 or float64.

Outputs:

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$. Has the data type of *x*.

Raises

- **TypeError** –If *kernel_size* or *strides* is neither an int nor a tuple.
- **TypeError** –If *pad_mode* or *data_format* is not a string.
- **ValueError** –If numbers in *kernel_size* or *strides* are not positive.
- **ValueError** –If *pad_mode* is not one of `"SAME"`, `"VALID"` or `"PAD"`.
- **ValueError** –If *pad_mode* is `"SAME"` or `"VALID"`, *ceil_mode* is not `None`.
- **ValueError** –If *kernel_size* or *strides* is a tuple whose length is not equal to 3.
- **ValueError** –If *data_format* is not `"NCDHW"`.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(1 * 2 * 2 * 2 * 3).reshape((1, 2, 2, 2, 3)), mindspore.float32)
>>> max_pool3d = ops.MaxPool3D(kernel_size=2, strides=1, pad_mode="VALID")
>>> output = max_pool3d(x)
>>> print(output)
[[[[[10. 11.]]]
  [[[22. 23.]]]]]
```

mindspore.ops.MaxPool3DWithArgmax

class mindspore.ops.**MaxPool3DWithArgmax**(*ksize*, *strides*, *pads*, *dilation=(1, 1, 1)*, *ceil_mode=False*, *data_format='NCDHW'*, *argmax_type=mstype.int64*)

Performs a 3D max pooling on the input Tensor and returns both max values and indices.

Typically the input is a Tensor with shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$, outputs regional maximum in the (D_{in}, H_{in}, W_{in})-dimension. Given *ksize* $ks = (d_{ker}, h_{ker}, w_{ker})$ and *strides* $s = (s_0, s_1, s_2)$, the operation is as follows.

$$\text{output}(N_i, C_j, d, h, w) = \max_{l=0,\ldots,d_{ker}-1} \max_{m=0,\ldots,h_{ker}-1} \max_{n=0,\ldots,w_{ker}-1} \text{input}(N_i, C_j, s_0 \times d + l, s_1 \times h + m, s_2 \times w + n)$$

The output is a Tensor with shape $(N_{out}, C_{out}, D_{out}, H_{out}, W_{out})$ and its depth, height and width are:

$$\begin{aligned} D_{out} &= \frac{D_{in}+2\times\text{pads}[0]-\text{dilation}[0]\times(\text{ksize}[0]-1)-1}{\text{stride}[0]} + 1 \\ H_{out} &= \frac{H_{in}+2\times\text{pads}[1]-\text{dilation}[1]\times(\text{ksize}[1]-1)-1}{\text{stride}[1]} + 1 \\ W_{out} &= \frac{W_{in}+2\times\text{pads}[2]-\text{dilation}[2]\times(\text{ksize}[2]-1)-1}{\text{stride}[2]} + 1 \end{aligned}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **ksize** (*Union[int, tuple[int]]*) – The size of kernel used to take the maximum value and arg value, is an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively.
- **strides** (*Union[int, tuple[int]]*) – The distance of kernel moving, an int number that represents the depth, height and width of movement are both strides, or a tuple of three int numbers that represent depth, height and width of movement respectively.
- **pads** (*Union[int, tuple[int]]*) – An int number that represents the depth, height and width of movement are both strides, or a tuple of three int numbers that represent depth, height and width of movement respectively.
- **dilation** (*Union[int, tuple[int]]*) – Default: `(1, 1, 1)` .
- **ceil_mode** (*bool*) – Whether to use ceil instead of floor to calculate output shape. Default: `False` .
- **data_format** (*str*) – The optional value for data format. Currently only support `'NCDHW'` . Default: `'NCDHW'` .
- **argmax_type** (`mindspore.dtype`) – The dtype for argmax. Default: `mstype.int64` .

Inputs:

- **x** (Tensor) - Tensor of shape $(N_{in}, C_{in}, D_{in}, H_{in}, W_{in})$ with data type of int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32 or float64.

Outputs:

Tuple of 2 Tensors, representing the maxpool result and where the max values are generated.

- **output** (Tensor) - Maxpooling result, with shape $(N_{out}, C_{out}, D_{out}, H_{out}, W_{out})$. It has the same data type as *x*.
- **argmax** (Tensor) - Index corresponding to the maximum value. Data type is int32 or int64.

Raises

- **TypeError** – If *x* is not a Tensor.
- **ValueError** – If length of shape of *x* is not equal to 5.
- **TypeError** – If *ksize* , *strides* , *pads* or *dilation* is not int or tuple.
- **ValueError** – If *ksize* or *strides* is less than 1.
- **ValueError** – If *pads* is less than 0.
- **ValueError** – If *data_format* is not `'NCDHW'`.
- **ValueError** – If *argmax_type* is not mindspore.int64 or mindspore.int32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.arange(2 * 1 * 2 * 2 * 2).reshape((2, 1, 2, 2, 2)), mindspore.float32)
>>> max_pool3d_with_arg_op = ops.MaxPool3DWithArgmax(ksize=2, strides=1, pads=1)
>>> output_tensor, argmax = max_pool3d_with_arg_op(x)
>>> print(output_tensor.shape)
(2, 1, 3, 3, 3)
>>> print(argmax.shape)
(2, 1, 3, 3, 3)
```

mindspore.ops.MaxUnpool2D

class mindspore.ops.**MaxUnpool2D**(*ksize*, *strides=0*, *pads=0*, *output_shape=()*, *data_format='NCHW'*)

Calculates the partial inverse of MaxPool2D operation.

Since MaxPool2D loses non-maximal values, it is not fully invertible. Therefore, MaxUnpool2D takes the output of MaxPool2D, including the indices of the maximal values, and computes a partial inverse where all non-maximal values are set to zero. Typically the input is of shape (N, C, H_{in}, W_{in}) , the output is of shape (N, C, H_{out}, W_{out}) , the operation is as follows:

$$H_{out} = (Hin - 1) \times strides[0] - 2 \times pads[0] + ksize[0]$$
$$W_{out} = (Win - 1) \times strides[1] - 2 \times pads[1] + ksize[1]$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **ksize** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value, is an int number that represents height and width of the kernel, or a tuple of two int numbers that represent height and width respectively.
- **strides** (*Union[int, tuple[int]], optional*) –The strides of kernel moving. If *strides* is 0 or (0, 0), then *strides* equal to *ksize* . Default: `0` .
  - An int number that represents the height and width of movement are both *strides* .
  - A tuple of two int numbers that represent height and width of movement respectively.
- **pads** (*Union[int, tuple[int]], optional*) –The pad value to be filled. Default: `0` .
  - If *pads* is an integer, the paddings of height and width are the same, equal to pads.
  - If *pads* is a tuple of two integers, the padding of height and width equal to pads[0] and pads[1] correspondingly.
- **output_shape** (*tuple[int], optional*) –The target output size is an optional input. Default: `()` .
  - If $output\_shape == ()$, then the shape of output computed by *kszie*, *strides* and *pads* .
  - If $output\_shape! = ()$, then *output_shape* must be (N, C, H, W) or (N, H, W, C) and *output_shape* must belong to $[(N, C, H_{out} - strides[0], W_{out} - strides[1]), (N, C, H_{out} + strides[0], W_{out} + strides[1])]$.
- **data_format** (*str, optional*) –The optional value for data format. Currently support `"NCHW"` and `"NHWC"` . Default: `"NCHW"` .

Inputs:

- **x** (Tensor) - The input Tensor to invert. Tensor of shape (N, C, H_{in}, W_{in}) or (N, H_{in}, W_{in}, C).
- **argmax** (Tensor) - Max values' index represented by the *argmax*. Tensor of shape must be same with input *x*. Values of *argmax* must belong to $[0, H_{in} \times W_{in} - 1]$. Data type must be in int32 or int64.

Outputs:

Tensor, with shape (N, C, H_{out}, W_{out}) or (N, H_{out}, W_{out}, C). Has the same data type with *x*.

Raises

- **TypeError** –If data type of *x* or *argmax* is not supported.
- **TypeError** –If *ksize*, *strides* or *pads* is neither int nor tuple.
- **ValueError** –If numbers in *strides* (also support 0 and (0, 0)) or *ksize* is not positive.
- **ValueError** –If numbers in *pads* is negative.
- **ValueError** –If *ksize*, *strides* or *pads* is a tuple whose length is not equal to 2.
- **ValueError** –If *data_format* is not a str or is neither *NCHW* nor *NHWC*.
- **ValueError** –If *output_shape* whose length is neither 0 or 4.
- **ValueError** –If *output_shape* is not close to output size computed by attr *ksize*, *strides* and *pads*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[[0, 1], [8, 9]]]]).astype(np.float32))
>>> argmax = Tensor(np.array([[[[0, 1], [2, 3]]]]).astype(np.int64))
>>> maxunpool2d = ops.MaxUnpool2D(ksize=1, strides=1, pads=0)
>>> output = maxunpool2d(x, argmax)
>>> print(output.asnumpy())
[[[[0. 1.]
    [8. 9.]]]]
```

mindspore.ops.MaxUnpool3D

class mindspore.ops.**MaxUnpool3D**(*ksize*, *strides=0*, *pads=0*, *output_shape=()*, *data_format='NCDHW'*)

Computes the inverse of *mindspore.ops.MaxPool3D*.

MaxUnpool3D keeps the maximal value and set all position of non-maximal values to zero. Typically the input is of shape $(N, C, D_{in}, H_{in}, W_{in})$, the output is of shape $(N, C, D_{out}, H_{out}, W_{out})$, the operation is as follows.

$$
\begin{aligned}
D_{out} &= (Din - 1) \times strides[0] - 2 \times pads[0] + ksize[0] \\
H_{out} &= (Hin - 1) \times strides[1] - 2 \times pads[1] + ksize[1] \\
W_{out} &= (Win - 1) \times strides[2] - 2 \times pads[2] + ksize[2]
\end{aligned}
$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **ksize** (*Union[int, tuple[int]]*) –The size of kernel used to take the maximum value, is an int number that represents depth, height and width of the kernel, or a tuple of three int numbers that represent depth, height and width respectively.
- **strides** (*Union[int, tuple[int]], optional*) –The distance of kernel moving. Default: `0` .
  - If it is an int number, the depth, height and width of movement are all equal to *strides*.
  - If it is a tuple of three int numbers, they represent depth, height and width of movement respectively.
  - If strides is 0 or (0, 0, 0), then *strides* equal to *ksize*.
- **pads** (*Union[int, tuple[int]], optional*) –The pad value to be filled. Default: `0` .
  - If *pads* is an integer, the paddings of depth, height and width are the same, equal to pads.
  - If *pads* is a tuple of three integers, the padding of depth, height and width equal to pads[0], pads[1] and pads[2] correspondingly.
- **output_shape** (*tuple[int], optional*) –The target output size. Default: `()` . If $output\_shape == ()$, then the shape of output computed by kszie, strides and pads shown above. If $output\_shape != ()$, then output_shape format must be (N, C, D, H, W) or (N, D, H, W, C) and output_shape must be in range $[(N, C, D_{out} - strides[0], H_{out} - strides[1], W_{out} - strides[2]), (N, C, D_{out} + strides[0], H_{out} + strides[1], W_{out} + strides[2])]$.
- **data_format** (*str, optional*) –The optional value for data format. Currently support `'NCDHW'` and `'NDHWC'` . Default: `'NCDHW'` .

Inputs:

- **x** (Tensor) - The input Tensor to invert. Tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$ or $(N, D_{in}, H_{in}, W_{in}, C)$.
- **argmax** (Tensor) - Max values' index. Tensor that has the same shape as *x*. Values of *argmax* must be in range $[0, D_{in} \times H_{in} \times W_{in} - 1]$. Data type must be int32 or int64.

Outputs:

Tensor, with shape $(N, C, D_{out}, H_{out}, W_{out})$ or $(N, D_{out}, H_{out}, W_{out}, C)$. Has the same data type with *x*.

Raises

- **TypeError** –If data type of *x* or *argmax* is Number.
- **TypeError** –If *ksize*, *strides* or *pads* is neither int nor tuple.
- **ValueError** –If numbers in *strides* or *ksize* is negative.
- **ValueError** –If numbers in *pads* is negative.
- **ValueError** –If *ksize*, *strides* or *pads* is a tuple whose length is not equal to 3.
- **ValueError** –If *data_format* is not a str or is neither `'NCDHW'` nor `'NDHWC'`.
- **ValueError** –If *output_shape* whose length is neither 0 or 5.
- **ValueError** –If *output_shape* is not close to output size range computed by attr *ksize, strides, pads*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[[[0, 1], [8, 9]]]]]).astype(np.float32))
>>> argmax = Tensor(np.array([[[[[0, 1], [2, 3]]]]]).astype(np.int64))
>>> maxunpool3d = ops.MaxUnpool3D(ksize=1, strides=1, pads=0)
>>> output = maxunpool3d(x, argmax)
>>> print(output.asnumpy())
[[[[[0. 1.]
    [8. 9.]]]]]
```

mindspore.ops.MirrorPad

class mindspore.ops.**MirrorPad**(*mode='REFLECT'*)

Pads the input tensor according to the paddings and mode.

Parameters

mode (*str, optional*) –An optional string specifying the pad method. The optional values are `'REFLECT'` and `'SYMMETRIC'`. Default: `'REFLECT'`.

- `'REFLECT'`: Reflect the value on the edge while omitting the last one. For example, pad [1, 2, 3, 4] with 2 elements on both sides will result in [3, 2, 1, 2, 3, 4, 3, 2].
- `'SYMMETRIC'`: Reflect the value on the edge while repeating the last one. For example, pad [1, 2, 3, 4] with 2 elements on both sides will result in [2, 1, 1, 2, 3, 4, 4, 3].

Inputs:

- **input_x** (Tensor) - Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions.
- **paddings** (Tensor) - Paddings requires constant tensor. The value of *paddings* is a matrix(list), and its shape is $(N, 2)$. N is the rank of input data. All elements of paddings are int type. For the input in the *D* th dimension, paddings[D, 0] indicates how many sizes to be extended ahead of the input tensor in the *D* th dimension, and paddings[D, 1] indicates how many sizes to be extended behind the input tensor in the *D* th dimension. Both paddings[D, 0] and paddings[D, 1] must be no greater than input_x.dim_size(D) (or input_x.dim_size(D) - 1) if mode is SYMMETRIC (if REFLECT, respectively).

Outputs:

Tensor, the tensor after padding.

- If *mode* is `'REFLECT'`, it uses a way of symmetrical copying through the axis of symmetry to fill in. If the *input_x* is [[1,2,3], [4,5,6], [7,8,9]] and *paddings* is [[1,1], [2,2]], then the *Outputs* is [[6,5,4,5,6,5,4], [3,2,1,2,3,2,1], [6,5,4,5,6,5,4], [9,8,7,8,9,8,7], [6,5,4,5,6,5,4]]. For a more intuitive understanding, please see the example below.
- If *mode* is `'SYMMETRIC'`, the filling method is similar to the `'REFLECT'`. It is also copied according to the symmetry axis, except that it includes the symmetry axis. If the *input_x* is [[1,2,3], [4,5,6], [7,8,9]] and *paddings* is [[1,1], [2,2]], then the *Outputs* is [[2,1,1,2,3,3,2], [2,1,1,2,3,3,2], [5,4,4,5,6,6,5], [8,7,7,8,9,9,8], [8,7,7,8,9,9,8]]. For a more intuitive understanding, please see the example below.

Raises

- **TypeError** –If *input_x* or *paddings* is not a Tensor.
- **TypeError** –If *mode* is not a str.
- **ValueError** –If paddings.size is not equal to 2 * rank of input_x.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, nn, ops
>>> # case1: mode="REFLECT"
>>> class Net(nn.Cell):
...     def __init__(self, mode):
...         super(Net, self).__init__()
...         self.pad = ops.MirrorPad(mode=mode)
...         self.paddings = Tensor([[1, 1], [2, 2]])
...     def construct(self, input_x):
...         return self.pad(input_x, self.paddings)
...
>>> input_x = Tensor([[1,2,3], [4,5,6], [7,8,9]])
>>> pad = Net("REFLECT")
>>> output = pad(input_x)
>>> print(output)
[[6 5 4 5 6 5 4]
 [3 2 1 2 3 2 1]
 [6 5 4 5 6 5 4]
 [9 8 7 8 9 8 7]
 [6 5 4 5 6 5 4]]
>>> # case2: mode="SYMMETRIC"
>>> pad = Net("SYMMETRIC")
>>> output = pad(input_x)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[2 1 1 2 3 3 2]
 [2 1 1 2 3 3 2]
 [5 4 4 5 6 6 5]
 [8 7 7 8 9 9 8]
 [8 7 7 8 9 9 8]]
```

mindspore.ops.Pad

class mindspore.ops.**Pad**(*paddings*)

Pads the input tensor according to the paddings.

Refer to *mindspore.ops.pad()* for more details. Use *mindspore.ops.pad()* instead if *paddings* has negative values.

Parameters

paddings (*tuple*) – The shape of parameter *paddings* is (N, 2). N is the rank of input data. All elements of paddings are int type. For the input in *D* th dimension, paddings[D, 0] indicates how many sizes to be extended ahead of the input tensor in the *D* th dimension, and paddings[D, 1] indicates how many sizes to be extended behind the input tensor in the *D* th dimension.

Inputs:

- **input_x** (Tensor) - Tensor to be padded. It has shape $(N, *)$, where $*$ means any number of additional dimensions.

Outputs:

Tensor, the tensor after padding.

Raises

- **TypeError** – If *paddings* is not a tuple.
- **TypeError** – If *input_x* is not a Tensor.
- **ValueError** – If shape of *paddings* is not $(N, 2)$.
- **ValueError** – If paddings.size is not equal to 2 * len(input_x).

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> pad_op = ops.Pad(((1, 2), (2, 1)))
>>> output = pad_op(input_x)
>>> print(output)
[[ 0.   0.   0.   0.   0.   0. ]
 [ 0.   0.  -0.1  0.3  3.6  0. ]
 [ 0.   0.   0.4  0.5 -3.2  0. ]
 [ 0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0. ]]
```

mindspore.ops.EmbeddingLookup

class mindspore.ops.**EmbeddingLookup**

Returns a slice of input tensor based on the specified indices.

This Primitive has the similar functionality as GatherV2 operating on *axis = 0*, but has one more inputs: *offset*.

Inputs:

- **input_params** (Tensor) - a Tensor slice, the shape is $(x_1, x_2, ..., x_R)$. Currently, the dimension is restricted to be 2.
- **input_indices** (Tensor) - Specifies the indices of elements of the original Tensor. The shape is $(y_1, y_2, ..., y_S)$. Values can be out of range of *input_params*, and the exceeding part will be filled with 0 in the output. Values do not support negative and the result is undefined if values are negative. The data type should be int32 or int64.
- **offset** (int) - Specifies the offset value of this *input_params* slice. Thus the real indices are equal to *input_indices* minus *offset*.

Outputs:

Tensor, the shape of tensor is $(z_1, z_2, ..., z_N)$. The data type is the same with *input_params*.

Raises

- **TypeError** –If dtype of *input_indices* is not int.
- **ValueError** –If length of shape of *input_params* is greater than 2.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_params = Tensor(np.array([[8, 9], [10, 11], [12, 13], [14, 15]]), mindspore.
↪float32)
>>> input_indices = Tensor(np.array([[5, 2], [8, 5]]), mindspore.int32)
>>> offset = 4
>>> output = ops.EmbeddingLookup()(input_params, input_indices, offset)
>>> print(output)
[[[10. 11.]
  [ 0.  0.]]
 [[ 0.  0.]
  [10. 11.]]]
```

mindspore.ops.Padding

class mindspore.ops.**Padding**(*pad_dim_size=8*)

Extends the last dimension of the input tensor from 1 to pad_dim_size, by filling with 0.

Refer to *mindspore.ops.padding()* for more details.

Parameters

pad_dim_size (*int, optional*) – The value of the last dimension of *x* to be extended, which must be positive. Default: 8.

Inputs:

- **x** (Tensor) - Input Tensor of 2D or higher-dimensional. The last dimension of *x* must be 1. The data type is Number.

Outputs:

Tensor, the padded Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[8], [10]]), mindspore.float32)
>>> pad_dim_size = 4
>>> output = ops.Padding(pad_dim_size)(x)
>>> print(output)
[[ 8.  0.  0.  0.]
 [10.  0.  0.  0.]]
```

mindspore.ops.ResizeBicubic

class mindspore.ops.**ResizeBicubic**(*align_corners=False*, *half_pixel_centers=False*)

Resize images to size using bicubic interpolation.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **align_corners** (*bool, optional*) – If True, the centers of the 4 corner pixels of the input and output tensors are aligned, preserving the values at the corner pixels. Default: False.
- **half_pixel_centers** (*bool, optional*) – Whether to use half-pixel center alignment. If set to True, *align_corners* should be False. Default: False.

Inputs:

- **images** (Tensor) - The input image must be a 4-D tensor of shape $(batch, channels, height, width)$. The format must be NCHW. Types allowed: float16, float32, float64.

- **size** (Union[tuple[int], Tensor[int]]) - A 1-D tensor or tuple with 2 elements: new_height, new_width. Besides, tuple[int] is recommended.

Outputs:

A 4-D tensor with shape $(batch, channels, new\_height, new\_width)$ whose dtype is the same as *images* .

Raises

- **TypeError** –If the type of *images* is not allowed.
- **TypeError** –If the type of *align_corners* is not bool.
- **TypeError** –If the type of *half_pixel_centers* is not bool.
- **ValueError** –If the dim of *images* is not 4.
- **ValueError** –If the dim of *size* is not 1 when *size* is a tensor.
- **ValueError** –If the number of elements in *size* is not 2.
- **ValueError** –If any value of *size* is not positive.
- **ValueError** –If the values of *align_corners* and *half_pixel_centers* are both `True` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, nn
>>> class NetResizeBicubic(nn.Cell):
...     def __init__(self):
...         super(NetResizeBicubic, self).__init__()
...         align_corners = False
...         half_pixel_centers = False
...         self.resize = ops.ResizeBicubic(align_corners, half_pixel_centers)
...
...     def construct(self, images, size):
...         return self.resize(images, size)
...
>>> images = Tensor(np.array([1, 2, 3, 4]).reshape(1, 1, 2, 2).astype(np.float32))
>>> size = Tensor([1, 4], mindspore.int32)
>>> resizebicubic = NetResizeBicubic()
>>> output = resizebicubic(images, size)
>>> print(output)
    [[[[1. 1.5 2. 2.09375]]]]
```

mindspore.ops.ResizeNearestNeighbor

class mindspore.ops.**ResizeNearestNeighbor**(*size*, *align_corners=False*, *half_pixel_centers=False*)

Resizes the input tensor to a given size by using the nearest neighbor algorithm. The nearest neighbor algorithm selects the value of the nearest point and does not consider the values of neighboring points at all, yielding a piecewise-constant interpolant.

Parameters

- **size** (*Union[tuple, list]*) –The target size. The dimension of size must be 2.
- **align_corners** (*bool, optional*) –Whether the centers of the 4 corner pixels of the input and output tensors are aligned. Default: `False` .
- **half_pixel_centers** (*bool, optional*) –Whether half pixel center. If set to `True` , *align_corners* should be False. Default: `False` .

Inputs:

- **input_x** (Tensor) - The input tensor. The shape of the tensor is (N, C, H, W).

Outputs:

Tensor, the shape of the output tensor is (N, C, NEW_H, NEW_W). The data type is the same as the *input_x*.

Raises

- **TypeError** –If *size* is neither tuple nor list.
- **TypeError** –If *align_corners* is not a bool.
- **ValueError** –If length of *size* is not equal to 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input_tensor = Tensor(np.array([[[[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]]]), mindspore.
 ↪float32)
>>> size = (2, 2)
>>> output = ops.ResizeNearestNeighbor(size=size)(input_tensor)
>>> print(output)
[[[[-0.1  0.3]
   [ 0.4  0.5]]]]
```

18.2.2 Loss Function

API Name	Description	Supported Platforms
mindspore.ops.BCEWithLogitsLoss	Adds sigmoid activation function to *input* as logits, and uses the given logits to compute binary cross entropy between the logits and the target.	Ascend GPU CPU
mindspore.ops.BinaryCrossEntropy	Computes the binary cross entropy between the logits and the labels.	Ascend GPU CPU
mindspore.ops.CTCLoss	Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.	Ascend GPU CPU
mindspore.ops.CTCLossV2	Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.	Ascend GPU CPU
mindspore.ops.KLDivLoss	Computes the Kullback-Leibler divergence between the logits and the labels.	Ascend GPU CPU
mindspore.ops.L2Loss	Calculates half of the L2 norm, but do not square the result.	Ascend GPU CPU
mindspore.ops.MultilabelMarginLoss	Creates a loss criterion that minimizes the hinge loss for multi-class classification tasks.	Ascend GPU
mindspore.ops.MultiMarginLoss	Creates a loss function that minimizes the hinge loss for multi-class classification tasks.	Ascend GPU CPU
mindspore.ops.NLLLoss	Gets the negative log likelihood loss between logits and labels.	Ascend GPU CPU
mindspore.ops.RNNTLoss	Computes the RNNTLoss and its gradient with respect to the softmax outputs.	Ascend
mindspore.ops.SigmoidCrossEntropyWithLogits	Uses the given logits to compute sigmoid cross entropy between the logits and the label.	Ascend GPU CPU
mindspore.ops.SmoothL1Loss	Calculate the smooth L1 loss, and the L1 loss function has robustness.	Ascend GPU CPU
mindspore.ops.SoftMarginLoss	Refer to *mindspore.ops.soft_margin_loss()* for more details.	Ascend GPU
mindspore.ops.SoftmaxCrossEntropyWithLogits	Gets the softmax cross-entropy value between logits and labels with one-hot encoding.	Ascend GPU CPU
mindspore.ops.SparseSoftmaxCrossEntropyWithLogits	Computes the softmax cross-entropy value between logits and sparse encoding labels.	GPU CPU
mindspore.ops.TripletMarginLoss	TripletMarginLoss operation.	GPU

mindspore.ops.BCEWithLogitsLoss

class mindspore.ops.**BCEWithLogitsLoss**(*reduction='mean'*)

Adds sigmoid activation function to *input* as logits, and uses the given logits to compute binary cross entropy between the logits and the target.

Sets input *input* as X, input *target* as Y, input weight as W, output as L. Then,

$$p_{ij} = sigmoid(X_{ij}) = \frac{1}{1+e^{-X_{ij}}}$$
$$L_{ij} = -[Y_{ij}log(p_{ij}) + (1 - Y_{ij})log(1 - p_{ij})]$$

i indicates the i^{th} sample, j indicates the category. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

ℓ indicates the method of calculating the loss. There are three methods: the first method is to provide the loss value directly, the second method is to calculate the average value of all losses, and the third method is to calculate the sum of all losses.

This operator will multiply the output by the corresponding weight. The tensor *weight* assigns different weights to each piece of data in the batch, and the tensor *pos_weight* adds corresponding weights to the positive examples of each category.

In addition, it can trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$p_{ij,c} = sigmoid(X_{ij,c}) = \frac{1}{1+e^{-X_{ij,c}}}$$
$$L_{ij,c} = -[P_c Y_{ij,c} * log(p_{ij,c}) + (1 - Y_{ij,c}) log(1 - p_{ij,c})]$$

where c is the class number (c>1 for multi-label binary classification, c=1 for single-label binary classification), n is the number of the sample in the batch and P_c is the weight of the positive answer for the class c. $P_c > 1$ increases the recall, $P_c < 1$ increases the precision.

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the weighted mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **input** (Tensor) - Input *input*. Tensor of shape $(N, *)$ where $*$ means, any number of additional dimensions. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **target** (Tensor) - Ground truth label, has the same shape as *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **weight** (Tensor) - A rescaling weight applied to the loss of each batch element. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).
- **pos_weight** (Tensor) - A weight of positive examples. Must be a vector with length equal to the number of classes. It can be broadcast to a tensor with shape of *input*. Data type must be float16, float32 or bfloat16(only Atlas A2 series products are supported).

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, it's a tensor with the same shape and type as input *input*. Otherwise, the output is a scalar.

Raises

- **TypeError** – If any input is not Tensor.
- **TypeError** – If data type of any input is not float16, float32 or bfloat16.
- **TypeError** – If data type of *reduction* is not string.
- **ValueError** – If *weight* or *pos_weight* can not be broadcast to a tensor with shape of *input*.
- **ValueError** – If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]), mindspore.float32)
>>> target = Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]), mindspore.float32)
>>> weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> pos_weight = Tensor(np.array([1.0, 1.0, 1.0]), mindspore.float32)
>>> loss = ops.BCEWithLogitsLoss()
>>> output = loss(input, target, weight, pos_weight)
>>> print(output)
0.3463612
```

mindspore.ops.BinaryCrossEntropy

class mindspore.ops.**BinaryCrossEntropy**(*reduction='mean'*)

Computes the binary cross entropy between the logits and the labels.

Sets logits as x, labels as y, output as $\ell(x, y)$. Let,

$$L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

In which, L indicates the loss of all batch_sizes, l indicates the loss of one batch_size, and n indicates one batch_size in the 1-N range, w_n indicates the weight of n-th batch of binary cross entropy. Then,

$$\ell(x, y) = \begin{cases} L, & \text{if reduction = 'none';} \\ \text{mean}(L), & \text{if reduction = 'mean';} \\ \text{sum}(L), & \text{if reduction = 'sum'.} \end{cases}$$

Warning:

- The value of x must range from 0 to 1.

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the weighted mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - The predictive value whose data type must be float16 or float32, The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **labels** (Tensor) - The target value which has the same shape and data type as *logits*. And the data type is float16 or float32.
- **weight** (Tensor, optional) - A rescaling weight applied to the loss of each batch element. And it must have the same shape and data type as *logits*. Default: `None` .

Outputs:

Tensor or Scalar. Returns Tensor that has the same dtype and shape as *logits* if *reduction* is 'none'. Otherwise, returns a scalar Tensor.

Raises

- **TypeError** –If dtype of *logits*, *labels* or *weight* (if given) is neither float16 nor float32.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'` or `'sum'`.
- **ValueError** –If shape of *labels* is not the same as *logits* or *weight* (if given).
- **TypeError** –If *logits*, *labels* or *weight* is not a Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.binary_cross_entropy = ops.BinaryCrossEntropy()
...     def construct(self, logits, labels, weight):
...         result = self.binary_cross_entropy(logits, labels, weight)
...         return result
...
>>> net = Net()
>>> logits = Tensor(np.array([0.2, 0.7, 0.1]), mindspore.float32)
>>> labels = Tensor(np.array([0., 1., 0.]), mindspore.float32)
>>> weight = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = net(logits, labels, weight)
>>> print(output)
0.38240486
```

mindspore.ops.CTCLoss

class mindspore.ops.**CTCLoss**(*preprocess_collapse_repeated=False*, *ctc_merge_repeated=True*, *ignore_longer_outputs_than_inputs=False*)

Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.

The bottom layer of this interface calls the implementation of the third-party baidu-research::warp-ctc. The CTC algorithm is proposed in Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks.

CTCLoss calculates loss between a continuous time series and a target sequence. CTCLoss sums over the probability of input to target, producing a loss value which is differentiable with respect to each input node. The alignment of input to target is assumed to be "many-to-one", such that the length of target series must be less than or equal to the length of input.

Parameters

- **preprocess_collapse_repeated** (*bool, optional*) –If `True`, repeated labels will be collapsed prior to the CTC calculation. Default: `False`.

- **ctc_merge_repeated** (*bool, optional*) –If `False` , during CTC calculation, repeated non-blank labels will not be merged and these labels will be interpreted as individual ones. This is a simplified version of CTC. Default: `True` .
- **ignore_longer_outputs_than_inputs** (*bool, optional*) –If `True` , sequences with longer outputs than inputs will be ignored. Default: `False` .

Inputs:

- **x** (Tensor) - The input Tensor must be a *3-D* tensor whose shape is $(max\_time, batch\_size, num\_classes)$. *num_classes* must be *num_labels + 1* classes, *num_labels* indicates the number of actual labels. Blank labels are reserved. Default blank label is *num_classes - 1*. Data type must be float16, float32 or float64.
- **labels_indices** (Tensor) - The indices of labels. *labels_indices[i, :] = [b, t]* means *labels_values[i]* stores the id for *(batch b, time t)*. The type must be int64 and rank must be 2.
- **labels_values** (Tensor) - A *1-D* input tensor. The values are associated with the given batch and time. The type must be int32. *labels_values[i]* must be in the range of *[0, num_classes)*.
- **sequence_length** (Tensor) - A tensor containing sequence lengths with the shape of $(batch\_size,)$. The type must be int32. Each value in the tensor must not be greater than *max_time*.

Outputs:

- **loss** (Tensor) - A tensor containing log-probabilities, the shape is $(batch\_size,)$. The tensor has the same data type as *x*.
- **gradient** (Tensor) - The gradient of *loss*, has the same shape and data type as *x*.

Raises

- **TypeError** –If *preprocess_collapse_repeated*, *ctc_merge_repeated* or *ignore_longer_outputs_than_inputs* is not a bool.
- **TypeError** –If *x*, *labels_indices*, *labels_values* or *sequence_length* is not a Tensor.
- **ValueError** –If rank of *labels_indices* is not equal to 2.
- **TypeError** –If dtype of *x* is not one of the following: float16, float32 nor float64.
- **TypeError** –If dtype of *labels_indices* is not int64.
- **TypeError** –If dtype of *labels_values* or *sequence_length* is not int32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[0.3, 0.6, 0.6],
...                       [0.4, 0.3, 0.9]],
...
...                      [[0.9, 0.4, 0.2],
...                       [0.9, 0.9, 0.1]]]).astype(np.float32))
>>> labels_indices = Tensor(np.array([[0, 0], [1, 0]]), mindspore.int64)
>>> labels_values = Tensor(np.array([2, 2]), mindspore.int32)
>>> sequence_length = Tensor(np.array([2, 2]), mindspore.int32)
```

(continues on next page)

(continued from previous page)

```
>>> ctc_loss = ops.CTCLoss()
>>> loss, gradient = ctc_loss(x, labels_indices, labels_values, sequence_length)
>>> print(loss)
[ 0.79628  0.5995158 ]
>>> print(gradient)
[[[ 0.27029088  0.36485454  -0.6351454  ]
  [ 0.28140804  0.25462854  -0.5360366 ]]
 [[ 0.47548494  0.2883962    0.04510255 ]
  [ 0.4082751   0.4082751    0.02843709 ]]]
```

mindspore.ops.CTCLossV2

class mindspore.ops.**CTCLossV2**(*blank=0*, *reduction='none'*, *zero_infinity=False*)

Calculates the CTC (Connectionist Temporal Classification) loss and the gradient.

The CTC algorithm is proposed in Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **blank** (*int, optional*) – The blank label. Default: `0` .
- **reduction** (*str, optional*) – Apply specific reduction method to the output. Currently only support `'none'`. Default: `'none'` .
- **zero_infinity** (*bool, optional*) – If loss is infinite, this parameter determines whether to set that loss and its correlated gradient to zero. Default: `False` .

Inputs:

- **log_probs** (Tensor) - A 3D tensor of shape (T, N, C), where T is input length, N is batch size and C is number of classes (including blank). Supported dtypes: float32, float64.
- **targets** (Tensor) - A 2D tensor of shape (N, S), where S is max target length, means the target sequences. Supported dtypes: int32, int64.
- **input_lengths** (Union(Tuple, Tensor)) - A tuple or Tensor of shape (N). It means the lengths of the input. Supported dtypes: int32, int64.
- **target_lengths** (Union(Tuple, Tensor)) - A tuple or Tensor of shape (N). It means the lengths of the target. Supported dtypes: int32, int64.

Outputs:

- **neg_log_likelihood** (Tensor) - A loss value which is differentiable with respect to each input node.
- **log_alpha** (Tensor) - The probability of possible trace of input to target.

Raises

- **TypeError** – If *zero_infinity* is not a bool.
- **TypeError** – If *reduction* is not string.
- **TypeError** – If the dtype of *log_probs* is not float or double.

- **TypeError** –If the dtype of *targets*, *input_lengths* or *target_lengths* is not int32 or int64.
- **ValueError** –If the rank of *log_probs* is not 3.
- **ValueError** –If the rank of *targets* is not 2.
- **ValueError** –If the shape of *input_lengths* does not match batch_size *N*.
- **ValueError** –If the shape of *target_lengths* does not match batch_size *N*.
- **TypeError** –If the types of *targets*, *input_lengths* or *target_lengths* are different.
- **ValueError** –If the value of *blank* is not in range [0, C).
- **RuntimeError** –If any value of *input_lengths* is larger than (num_labels|C).
- **RuntimeError** –If any *target_lengths[i]* is not in range [0, *input_length[i]*].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> log_probs = Tensor(np.array([[[0.3, 0.6, 0.6]],
...                              [[0.9, 0.4, 0.2]]]).astype(np.float32))
>>> targets = Tensor(np.array([[0, 1]]), mstype.int32)
>>> input_lengths = Tensor(np.array([2]), mstype.int32)
>>> target_lengths = Tensor(np.array([1]), mstype.int32)
>>> CTCLossV2 = ops.CTCLossV2(blank=0, reduction='none', zero_infinity=False)
>>> neg_log_hood, log_alpha = CTCLossV2(
...     log_probs, targets, input_lengths, target_lengths)
>>> print(neg_log_hood)
[-2.2986124]
>>> print(log_alpha)
[[[0.3       0.3           -inf      -inf      -inf]
  [1.2       1.8931472 1.2           -inf      -inf]]]
```

mindspore.ops.KLDivLoss

class mindspore.ops.**KLDivLoss**(*reduction='mean'*)

Computes the Kullback-Leibler divergence between the logits and the labels.

For tensors of the same shape x and $target$, the updating formulas of KLDivLoss algorithm are as follows,

$$L(x, target) = target \cdot (\log target - x)$$

Then,

$$\ell(x, target) = \begin{cases} L(x, target), & \text{if reduction = 'none';} \\ \text{mean}(L(x, target)), & \text{if reduction = 'mean';} \\ \text{sum}(L(x, target))/x.\text{shape}[0], & \text{if reduction = 'batchmean';} \\ \text{sum}(L(x, target)), & \text{if reduction = 'sum'.} \end{cases}$$

where x represents *logits*, $target$ represents *labels*, and $\ell(x, target)$ represents *output*.

Note:

- On Ascend, float64 dtype is not currently supported.
- The output aligns with the mathematical definition of Kullback-Leibler divergence only when *reduction* is set to `'batchmean'`.
- On Ascend, the value of *reduction* must be one of `'batchmean'`, `'none'` or `'sum'`.
- On GPU, the value of *reduction* must be one of `'mean'`, `'none'` or `'sum'`.
- On CPU, the value of *reduction* must be one of `'mean'`, `'batchmean'`, `'none'` or `'sum'`.

Parameters

reduction (*str, optional*) – Specifies the reduction to be applied to the output. Default: `'mean'` .

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.
- `'batchmean'`: average loss is taken over the batch, similar to the mean mode.

Inputs:

- **logits** (Tensor) - The input Tensor. The data type must be float16, float32 or float64.
- **labels** (Tensor) - The label Tensor which has the same shape and data type as *logits*.

Outputs:

Tensor or Scalar, if *reduction* is `'none'`, then output is a tensor and has the same shape as *logits*. Otherwise it is a scalar.

Raises

- **TypeError** – If *reduction* is not a str.
- **TypeError** – If neither *logits* nor *labels* is a Tensor.
- **TypeError** – If dtype of *logits* or *labels* is not currently supported.
- **ValueError** – If shape of *logits* is not the same as *labels*.
- **RuntimeError** – If *logits* or *labels* is a scalar when *reduction* is `'batchmean'`.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.kldiv_loss = ops.KLDivLoss(reduction='sum')
...     def construct(self, logits, labels):
```

(continues on next page)

(continued from previous page)

```
...         result = self.kldiv_loss(logits, labels)
...         return result
...
>>> net = Net()
>>> logits = Tensor(np.array([0.2, 0.7, 0.1]), mindspore.float32)
>>> labels = Tensor(np.array([0., 1., 0.]), mindspore.float32)
>>> output = net(logits, labels)
>>> print(output)
-0.7
```

mindspore.ops.L2Loss

class mindspore.ops.**L2Loss**

Calculates half of the L2 norm, but do not square the result.

Set input as x and output as loss.

$$loss = \frac{\sum x^2}{2}$$

Inputs:

- **input_x** (Tensor) - Tensor for computing the L2 norm. Data type must be float16, float32 or float64.

Outputs:

Tensor, has a Scalar Tensor with the same data type as *input_x*.

Raises

- **TypeError** – If *input_x* is not a Tensor.
- **TypeError** – If dtype of *input_x* is not float16, float32 or float64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([1, 2, 3]), mindspore.float16)
>>> l2_loss = ops.L2Loss()
>>> output = l2_loss(input_x)
>>> print(output)
7.0
```

mindspore.ops.MultilabelMarginLoss

class mindspore.ops.**MultilabelMarginLoss**(*reduction='mean'*)

Creates a loss criterion that minimizes the hinge loss for multi-class classification tasks. It takes a 2D mini-batch Tensor *x* as input and a 2D Tensor *y* containing target class indices as output.

Refer to *mindspore.ops.multilabel_margin_loss()* for more details.

Parameters

reduction (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.

- `'none'`: no reduction will be applied.
- `'mean'`: compute and return the mean of elements in the output.
- `'sum'`: the output elements will be summed.

Inputs:

- **x** (Tensor) - Predict data. Tensor of shape (C) or (N, C), where N is the batch size and C is the number of classes. Data type must be float16 or float32.
- **target** (Tensor) - Ground truth data, with the same shape as *input*, data type must be int32 and label targets padded by -1.

Outputs:

- **y** (Union[Tensor, Scalar]) - The loss of MultilabelMarginLoss. If *reduction* is `"none"`, its shape is (N). Otherwise, a scalar value will be returned.
- **is_target** (Tensor) - Output tensor for backward input, with the same shape as *target*, data type must be int32.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> loss = ops.MultilabelMarginLoss()
>>> x = Tensor(np.array([[0.1, 0.2, 0.4, 0.8], [0.2, 0.3, 0.5, 0.7]]), mindspore.float32)
>>> target = Tensor(np.array([[1, 2, 0, 3], [2, 3, -1, 1]]), mindspore.int32)
>>> output = loss(x, target)
>>> print(output)
(Tensor(shape=[], dtype=Float32, value= 0.325), Tensor(shape=[2, 4], dtype=Int32, value=
[[1, 1, 1, 1], [0, 0, 1, 1]]))
```

mindspore.ops.MultiMarginLoss

class mindspore.ops.**MultiMarginLoss**(*p=1*, *margin=1.0*, *reduction='mean'*)

Creates a loss function that minimizes the hinge loss for multi-class classification tasks. The loss is calculated by comparing the input and output of the function.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.multi_margin_loss()* for more details.

Parameters

- **p** (*int, optional*) – The norm degree for pairwise distance. Should be 1 or 2. Default: `1` .
- **margin** (*int, optional*) – A parameter to change pairwise distance. Default: `1.0` .
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **inputs** (Tensor) - Input , with shape (N, C). Data type only support float32, float16 or float64.
- **target** (Tensor) - Ground truth labels, with shape $(N,)$. Data type only support int64. The value of target should be non-negative, less than C.
- **weight** (Tensor, optional) - The rescaling weight to each class with shape $(C,)$. Data type only support float16, float32 or float64.

Outputs:

Tensor, When *reduction* is `'none'`, the shape is $(N,)$. Otherwise, it is a scalar. Has the same data type with *inputs*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.ones(shape=[3, 3]), mindspore.float32)
>>> target = Tensor(np.array([1, 2, 1]), mindspore.int64)
>>> weight = Tensor(np.array([1, 1, 1]), mindspore.float32)
>>> loss = ops.MultiMarginLoss()
>>> output = loss(x, target, weight)
>>> print(output)
0.6666667
```

mindspore.ops.NLLLoss

class mindspore.ops.**NLLLoss**(*reduction='mean'*, *ignore_index=- 100*)

Gets the negative log likelihood loss between logits and labels.

The nll loss with $reduction = none$ can be described as:

$$\ell(x, t) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{t_n} x_{n,t_n}, \quad w_c = \text{weight}\,[c] \cdot 1$$

where x is the logits, t is the labels, w is the weight, N is the batch size, c belonging to [0, C-1] is class index, where C is the number of classes.

If $reduction \neq none$ (default `'mean'`), then

$$\ell(x, t) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{tn}} l_n, & \text{if reduction = 'mean';} \\ \sum_{n=1}^{N} l_n, & \text{if reduction = 'sum'} \end{cases}$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'` , `'mean'` , `'sum'` . Default: `'mean'` .
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the weighted mean of elements in the output.
  - `'sum'`: the output elements will be summed.
- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. Default: `-100` .

Inputs:

- **logits** (Tensor) - Input logits, with shape (N, C). Data type only supports float32 or float16 or bfloat16(only supported by Atlas A2 training series products).
- **labels** (Tensor) - Ground truth labels, with shape $(N,)$, where each value belong to $[0, C-1]$. Data type only supports uint8 or int32 or int64.
- **weight** (Tensor) - The rescaling weight to each class, with shape $(C,)$ and data type only supports float32 or float16 or bfloat16(only supported by Atlas A2 training series products). It should have the same data type as *logits* .

Returns

Tuple of 2 tensors composed with *loss* and *total_weight*.

- **loss** (Tensor) - When *reduction* is `'none'` and *logits* is a 2D tensor, the *loss* shape is $(N,)$. Otherwise, the *loss* is a scalar. The data type is the same with that of *logits*.
- **total_weight** (Tensor) - The *total_weight* is a scalar. The data type is the same with that of *weight*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([[0.5488135, 0.71518934],
...                           [0.60276335, 0.5448832],
...                           [0.4236548, 0.6458941]]).astype(np.float32))
>>> labels = Tensor(np.array([0, 0, 0]).astype(np.int32))
>>> weight = Tensor(np.array([0.3834415, 0.79172504]).astype(np.float32))
>>> nll_loss = ops.NLLLoss(reduction="mean")
>>> loss, weight = nll_loss(logits, labels, weight)
>>> print(loss)
-0.52507716
>>> print(weight)
1.1503246
```

mindspore.ops.RNNTLoss

class mindspore.ops.**RNNTLoss**(*blank_label=0*)

Computes the RNNTLoss and its gradient with respect to the softmax outputs.

Parameters

blank_label (*int*) – blank label. Default: `0` .

Inputs:

- **acts** (Tensor) - Tensor of shape (B, T, U, V), where B is batch, T is sequence length, U is label length and V is output dim. Data type must be float16 or float32.
- **labels** (Tensor) - Tensor of shape $(B, U-1)$. Data type is int32.
- **input_lengths** (Tensor) - Tensor of shape $(B,)$. Data type is int32.
- **label_lengths** (Tensor) - Tensor of shape $(B,)$. Data type is int32.

Outputs:

- **costs** (Tensor) - Tensor of shape $(B,)$. Data type is int32.
- **grads** (Tensor) - Has the same shape and dtype as *acts*.

Raises

- **TypeError** – If *acts*, *labels*, *input_lengths* or *label_lengths* is not a Tensor.
- **TypeError** – If dtype of *acts* is neither float16 nor float32.
- **TypeError** – If dtype of *labels*, *input_lengths* or *label_lengths* is not int32.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import ops, Tensor
>>> B, T, U, V = 1, 2, 3, 5
>>> blank = 0
>>> acts = np.random.random((B, T, U, V)).astype(np.float32)
>>> labels = np.array([[1, 2]]).astype(np.int32)
>>> input_length = np.array([T] * B).astype(np.int32)
>>> label_length = np.array([len(l) for l in labels]).astype(np.int32)
>>> rnnt_loss = ops.RNNTLoss(blank_label=0)
>>> costs, grads = rnnt_loss(Tensor(acts), Tensor(labels), Tensor(input_length),␣
↪Tensor(label_length))
>>> print(costs.shape)
(1,)
>>> print(grads.shape)
(1, 2, 3, 5)
```

mindspore.ops.SigmoidCrossEntropyWithLogits

class mindspore.ops.**SigmoidCrossEntropyWithLogits**

Uses the given logits to compute sigmoid cross entropy between the logits and the label.

Measures the distribution error in discrete classification tasks where each class is independent and not mutually exclusive using cross entropy loss.

Sets input logits as X, input label as Y, output as $loss$. Then,

$$
\begin{aligned}
p_{ij} &= sigmoid(X_{ij}) = \frac{1}{1+e^{-X_{ij}}} \\
loss_{ij} &= -[Y_{ij} * ln(p_{ij}) + (1 - Y_{ij})ln(1 - p_{ij})]
\end{aligned}
$$

Inputs:

- **logits** (Tensor) - Input logits. Tensor of shape $(N, *)$ where $*$ means any number of additional dimensions.
- **label** (Tensor) - Ground truth label. With the same shape and type as *logits*.

Outputs:

Tensor, with the same shape and type as input *logits*.

Raises

TypeError – If *logits* or *label* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> logits = Tensor(np.array([[-0.8, 1.2, 0.7], [-0.1, -0.4, 0.7]]).astype(np.float32))
>>> labels = Tensor(np.array([[0.3, 0.8, 1.2], [-0.6, 0.1, 2.2]]).astype(np.float32))
>>> sigmoid = ops.SigmoidCrossEntropyWithLogits()
>>> output = sigmoid(logits, labels)
>>> print(output)
[[ 0.6111007   0.5032824   0.26318604]
 [ 0.58439666  0.5530153  -0.4368139 ]]
```

mindspore.ops.SmoothL1Loss

class mindspore.ops.**SmoothL1Loss**(*beta=1.0*, *reduction='none'*)

Calculate the smooth L1 loss, and the L1 loss function has robustness.

Refer to *mindspore.ops.smooth_l1_loss()* for more details.

> **Warning:** This API has poor performance on CPU and it is recommended to run it on the Ascend/GPU.

Parameters

- **beta** (*number, optional*) – A parameter used to control the point where the function will change between L1 to L2 loss. Default: `1.0`.
  - Ascend: The value should be equal to or greater than zero.
  - CPU/GPU: The value should be greater than zero.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'none'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **logits** (Tensor) - Input Tensor of any dimension. Supported dtypes:
  - Ascend: float16, float32, bfloat16.
  - CPU/GPU: float16, float32, float64.
- **labels** (Tensor) - Ground truth data.
  - CPU/Ascend: has the same shape as the *logits*, *logits* and *labels* comply with the implicit type conversion rules to make the data types consistent.
  - GPU: has the same shape and dtype as the *logits*.

Outputs:

Tensor, if *reduction* is `'none'`, then output is a tensor with the same shape as *logits*. Otherwise the shape of output tensor is ().

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> loss = ops.SmoothL1Loss()
>>> logits = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> labels = Tensor(np.array([1, 2, 2]), mindspore.float32)
>>> output = loss(logits, labels)
>>> print(output)
[0.  0.  0.5]
```

mindspore.ops.SoftMarginLoss

class mindspore.ops.**SoftMarginLoss**(*reduction='mean'*)

```
prim = ops.SoftMarginLoss(reduction)
out = prim(input, target)
```

is equivalent to

```
ops.soft_margin_loss(input, target, reduction)
```

Refer to *mindspore.ops.soft_margin_loss()* for more details.

mindspore.ops.SoftmaxCrossEntropyWithLogits

class mindspore.ops.**SoftmaxCrossEntropyWithLogits**

Gets the softmax cross-entropy value between logits and labels with one-hot encoding.

The updating formulas of SoftmaxCrossEntropyWithLogits algorithm are as follows,

$$p_{ij} = softmax(X_{ij}) = \frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)}$$
$$loss_{ij} = -\sum_j Y_{ij} * ln(p_{ij})$$

where X represents *logits*. Y represents *label*. $loss$ represents *output*.

Inputs:

- **logits** (Tensor) - Input logits, with shape (N, C). Data type must be float16 or float32.
- **labels** (Tensor) - Ground truth labels, with shape (N, C), has the same data type with *logits*.

Outputs:

Tuple of 2 tensors(*loss* , *dlogits*), the *loss* shape is $(N,)$, and the *dlogits* with the same shape as *logits*.

Raises

- **TypeError** –If dtype of *logits* or *labels* is neither float16 nor float32.
- **TypeError** –If *logits* or *labels* is not a Tensor.
- **ValueError** –If shape of *logits* is not the same as *labels*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> logits = Tensor([[2, 4, 1, 4, 5], [2, 1, 2, 4, 3]], mindspore.float32)
>>> labels = Tensor([[0, 0, 0, 0, 1], [0, 0, 0, 1, 0]], mindspore.float32)
>>> softmax_cross = ops.SoftmaxCrossEntropyWithLogits()
>>> loss, dlogits = softmax_cross(logits, labels)
>>> print(loss)
[0.5899297  0.52374405]
>>> print(dlogits)
[[ 0.02760027  0.20393994  0.01015357  0.20393994 -0.44563377]
 [ 0.08015892  0.02948882  0.08015892 -0.4077012   0.21789455]]
```

mindspore.ops.SparseSoftmaxCrossEntropyWithLogits

class mindspore.ops.**SparseSoftmaxCrossEntropyWithLogits**(*is_grad=False*)

Computes the softmax cross-entropy value between logits and sparse encoding labels.

Sets input logits as X, input label as Y, output as *loss*. The formula is as follows:

$$
\begin{aligned}
p_{ij} &= softmax(X_{ij}) = \frac{\exp(x_i)}{\sum_{j=0}^{N-1}\exp(x_j)} \\
loss_{ij} &= \begin{cases} -ln(p_{ij}), & j = y_i \\ 0, & j \neq y_i \end{cases} \\
loss &= \sum_{ij} loss_{ij}
\end{aligned}
$$

Parameters

is_grad (*bool, optional*) – If `True` , this operation returns the computed gradient. Default: `False` .

Inputs:

- **logits** (Tensor) - Input logits, with shape (N, C). Data type must be float16 or float32.
- **labels** (Tensor) - Ground truth labels, with shape (N). Data type must be int32 or int64.

Outputs:

Tensor, if *is_grad* is `False`, the output tensor is the value of loss; if *is_grad* is `True` , the output tensor is the gradient of input with the same shape as *logits*.

Raises

- **TypeError** – If *is_grad* is not a bool.
- **TypeError** – If dtype of *logits* is neither float16 nor float32.
- **TypeError** – If dtype of *labels* is neither int32 nor int64.
- **ValueError** – If $logits.shape[0] != labels.shape[0]$.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> logits = Tensor([[2, 3, 1, 4, 5], [2, 1, 2, 4, 3]], mindspore.float32)
>>> labels = Tensor([0, 1], mindspore.int32)
>>> sparse_softmax_cross = ops.SparseSoftmaxCrossEntropyWithLogits()
>>> loss = sparse_softmax_cross(logits, labels)
>>> print(loss)
3.4878292
>>> sparse_softmax_cross_grad = ops.SparseSoftmaxCrossEntropyWithLogits(is_grad=True)
>>> loss_grad = sparse_softmax_cross_grad(logits, labels)
>>> print(loss_grad)
[[-0.48415753  0.04306427  0.00582811  0.11706084  0.3182043 ]
 [ 0.04007946 -0.4852556   0.04007946  0.2961494   0.10894729]]
```

mindspore.ops.TripletMarginLoss

class mindspore.ops.**TripletMarginLoss**(*p=2*, *swap=False*, *eps=1e-6*, *reduction='mean'*)

TripletMarginLoss operation.

Creates a criterion that measures the triplet loss given an input tensors $x1$, $x2$, $x3$ and a margin with a value greater than 0. This is used for measuring a relative similarity between samples. A triplet is composed by a, p and n (i.e., *anchor*, *positive examples* and *negative examples* respectively). The shapes of all input tensors should be (N, D).

The distance swap is described in detail in the paper Learning local feature descriptors with triplets and shallow convolutional neural networks by V. Balntas, E. Riba et al.

The loss function for each sample in the mini-batch is:

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$$

where

$$d(x_i, y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_p$$

Parameters

- **p** (*int, optional*) – The norm degree for pairwise distance. Default: `2`.
- **eps** (*float, optional*) – Default: `1e-6`.
- **swap** (*bool, optional*) – The distance swap. Default: `False`.
- **reduction** (*str, optional*) – Apply specific reduction method to the output: `'none'`, `'mean'`, `'sum'`. Default: `'mean'`.
  - `'none'`: no reduction will be applied.
  - `'mean'`: compute and return the mean of elements in the output.
  - `'sum'`: the output elements will be summed.

Inputs:

- **x** (Tensor) - A sample randomly selected from the training set. Data type must be BasicType.
- **positive** (Tensor) - A sample belonging to the same category as x, with the same type and shape as *x*.
- **negative** (Tensor) - A sample belonging to the different class from x, with the same type and shape as *x*.

- **margin** (Tensor) - Make a margin between the positive pair and the negative pair.

Outputs:

Union[Tensor, Scalar], if *reduction* is `"none"`, a Ten sor will be returned with a shape of (N). Otherwise, a scalar value will be returned.

Raises

- **TypeError** –If *x*, *positive*, *negative*, or *margin* is not a Tensor.
- **TypeError** –If dtype of *x*, *positive*, or *negative* is not BasicType.
- **TypeError** –If dtypes of *x*, *positive* and *negative* are not the same.
- **TypeError** –If *margin* is not float32.
- **TypeError** –If *p* is not an int.
- **TypeError** –If *eps* is not a float.
- **TypeError** –If *swap* is not a bool.
- **ValueError** –If dimensions of input *x*, *positive* and *negative* are less than or equal to 1 at the same time.
- **ValueError** –If the dimension of input *x* or *positive* or *negative* is bigger than or equal to 8.
- **ValueError** –If length of shape of *margin* is not 0.
- **ValueError** –If shapes of *x*, *positive* and *negative* cannot broadcast.
- **ValueError** –If *reduction* is not one of `'none'`, `'mean'`, `'sum'`.

Supported Platforms:

`GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> loss = ops.TripletMarginLoss()
>>> x = Tensor(np.array([[0.3, 0.7], [0.5, 0.5]]), mindspore.float32)
>>> positive = Tensor(np.array([[0.4, 0.6], [0.4, 0.6]]), mindspore.float32)
>>> negative = Tensor(np.array([[0.2, 0.9], [0.3, 0.7]]), mindspore.float32)
>>> margin = Tensor(1.0, mindspore.float32)
>>> output = loss(x, positive, negative, margin)
>>> print(output)
0.8881968
```

18.2.3 Activation Function

API Name	Description	Supported Platforms
mindspore.ops.CeLU	Refer to mindspore.ops.celu() for more details.	Ascend GPU CPU
mindspore.ops.Elu	Refer to mindspore.ops.elu() for more details.	Ascend GPU CPU
mindspore.ops.FastGeLU	Refer to mindspore.ops.fast_gelu() for more details.	Ascend GPU CPU
mindspore.ops.GeLU	Gaussian Error Linear Units activation function.	Ascend GPU CPU
mindspore.ops.GLU	Computes GLU (Gated Linear Unit activation function) of the input tensor.	Ascend CPU
mindspore.ops.HShrink	Refer to mindspore.ops.hardshrink() for more details.	Ascend GPU CPU
mindspore.ops.HSigmoid	Refer to mindspore.ops.hardsigmoid() for more details.	Ascend GPU CPU
mindspore.ops.HSwish	Refer to mindspore.ops.hardswish() for more details.	Ascend GPU CPU
mindspore.ops.LogSoftmax	Refer to mindspore.ops.log_softmax() for more details.	Ascend GPU CPU
mindspore.ops.Mish	Computes MISH(A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.PReLU	Refer to mindspore.ops.prelu() for more details.	Ascend GPU CPU
mindspore.ops.ReLU	Refer to mindspore.ops.relu() for more details.	Ascend GPU CPU
mindspore.ops.ReLU6	Refer to mindspore.ops.relu6() for more details.	Ascend GPU CPU
mindspore.ops.SeLU	Activation function SeLU (Scaled exponential Linear Unit).	Ascend GPU CPU
mindspore.ops.Sigmoid	Refer to mindspore.ops.sigmoid() for more details.	Ascend GPU CPU
mindspore.ops.Softmax	Applies the Softmax operation to the input tensor on the specified axis.	Ascend GPU CPU
mindspore.ops.Softplus	Softplus activation function.	Ascend GPU CPU
mindspore.ops.SoftShrink	Refer to mindspore.ops.softshrink() for more details.	Ascend GPU CPU
mindspore.ops.Softsign	Softsign activation function.	Ascend GPU CPU
mindspore.ops.Tanh	Refer to mindspore.ops.tanh() for more details.	Ascend GPU CPU

mindspore.ops.CeLU

class mindspore.ops.**CeLU**(*alpha=1.0*)

```
prim = ops.CeLU(alpha)
out = prim(x)
```

is equivalent to

```
ops.celu(x, alpha)
```

Refer to *mindspore.ops.celu()* for more details.

mindspore.ops.Elu

class mindspore.ops.**Elu**(*alpha=1.0*)

```
prim = ops.Elu(alpha)
out = prim(input_x)
```

is equivalent to

```
ops.elu(input_x, alpha)
```

Refer to *mindspore.ops.elu()* for more details.

mindspore.ops.FastGeLU

class mindspore.ops.**FastGeLU**

```
prim = ops.FastGeLU()
out = prim(x)
```

is equivalent to

```
ops.fast_gelu(x)
```

Refer to *mindspore.ops.fast_gelu()* for more details.

mindspore.ops.GeLU

class mindspore.ops.**GeLU**

Gaussian Error Linear Units activation function.

GeLU is described in the paper Gaussian Error Linear Units (GELUs). Or refer to BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding for more details.

GeLU is defined as follows:

$$GELU(x_i) = x_i * P(X < x_i)$$

where P is the cumulative distribution function of the standard Gaussian distribution, x_i is the input element.

Note: When calculating the input gradient of GELU with an input value of infinity, there are differences in the output of the backward between 'Ascend' and 'GPU'. when x is -inf, the computation result of 'Ascend' is 0, and the computation result of 'GPU'

is Nan. when x is inf, the computation result of 'Ascend' is dy, and the computation result of 'GPU' is Nan. In mathematical terms, Ascend's result has higher precision.

Inputs:

- **x** (Tensor) - The input of the activation function GeLU, the data type is float16, float32 or float64.

Outputs:

Tensor, with the same type and shape as *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If dtype of *x* is not float16, float32 or float64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> result = ops.GeLU()(x)
>>> print(result)
[0.841192  1.9545976  2.9963627]
```

mindspore.ops.GLU

class mindspore.ops.**GLU**(*axis=-1*)

Computes GLU (Gated Linear Unit activation function) of the input tensor.

$$GLU(a, b) = a \otimes \sigma(b)$$

where a is the first half of the x Tensor after x is split and b is the second half.

Here σ is the sigmoid function, and $\otimes$ is the Hadamard product. See Language Modeling with Gated Convluational Networks .

Parameters

axis (*int, optional*) – Axis to split the input *x*. The value range is *[-r, r)* where *r* is the number of dimensions of *x*. Default: `-1` , the last dimension in *x*.

Inputs:

- **x** (Tensor) - Tensor to be calculated. Dtype is floating point and the shape is $(*_1, N, *_2)$ where * means, any number of additional dimensions. N is required to be an even number, where N is the size of x on the dimension selected by *axis*.

Outputs:

Tensor, the same dtype as *x*, with the shape $(*_1, M, *_2)$ where $M = N/2$.

Raises

- **TypeError** –If *x* is not a Tensor or *axis* is not an int.
- **IndexError** –If the value of *axis* is out of the range of *[-r, r)*, where *r* is the number of dimensions of *x*.
- **RuntimeError** –If dtype of *x* is not supported.
- **RuntimeError** –If the length of *x* in the dimension selected by *axis* is not even.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> from mindspore import ops, Tensor
>>> from mindspore import dtype as mstype
>>> import numpy as np
>>> axis = 0
>>> x = Tensor(np.array([0.3220, 0.9545, 0.7879, 0.0975, 0.3698,
...                      0.5135, 0.5740, 0.3435, 0.1895, 0.8764,
...                      0.4980, 0.9673, 0.9879, 0.6988, 0.9022,
...                      0.9304, 0.1558, 0.0153, 0.1559, 0.9852]).reshape([2, 2, 5]),␣
↪mstype.float32)
>>> glu = ops.GLU(axis=axis)
>>> y = glu(x)
>>> print(y)
[[[0.20028052 0.6916126  0.57412136 0.06512236 0.26307625]
  [0.3682598  0.3093122  0.17306386 0.10212085 0.63814086]]]
```

mindspore.ops.HShrink

class mindspore.ops.**HShrink**(*lambd=0.5*)

```
prim = ops.HShrink(lambd)
out = prim(input)
```

is equivalent to

```
ops.hardshrink(input, lambd)
```

Refer to *mindspore.ops.hardshrink()* for more details.

mindspore.ops.HSigmoid

class mindspore.ops.**HSigmoid**

```
prim = ops.HSigmoid()
out = prim(input)
```

is equivalent to

```
ops.hardsigmoid(input)
```

Refer to *mindspore.ops.hardsigmoid()* for more details.

mindspore.ops.HSwish

class mindspore.ops.**HSwish**

```
prim = ops.HSwish()
out = prim(input)
```

is equivalent to

```
ops.hardswish(input)
```

Refer to *mindspore.ops.hardswish()* for more details.

mindspore.ops.LogSoftmax

class mindspore.ops.**LogSoftmax**(*axis=-1*)

```
prim = ops.LogSoftmax(axis)
out = prim(logits)
```

is equivalent to

```
ops.log_softmax(logits, axis)
```

Refer to *mindspore.ops.log_softmax()* for more details.

mindspore.ops.Mish

class mindspore.ops.**Mish**

Computes MISH(A Self Regularized Non-Monotonic Neural Activation Function) of input tensors element-wise.

Refer to *mindspore.ops.mish()* for more details.

Inputs:

- **x** (Tensor) - The input Tensor. Supported dtypes:
  - GPU/CPU: float16, float32, float64.
  - Ascend: float16, float32.

Outputs:

Tensor, with the same type and shape as the *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> mish = ops.Mish()
>>> output = mish(x)
>>> print(output.shape)
(2, 3)
>>> x = Tensor(2.1, mindspore.float32)
>>> output = mish(x)
>>> print(output)
2.050599
```

mindspore.ops.PReLU

class mindspore.ops.**PReLU**

```
prim = ops.PReLU()
out = prim(input, weight)
```

is equivalent to

```
ops.prelu(input, weight)
```

Refer to *mindspore.ops.prelu()* for more details.

mindspore.ops.ReLU

class mindspore.ops.**ReLU**

```
prim = ops.ReLU()
out = prim(input)
```

is equivalent to

```
ops.relu(input)
```

Refer to *mindspore.ops.relu()* for more details.

mindspore.ops.ReLU6

class mindspore.ops.**ReLU6**

```
prim = ops.ReLU6()
out = prim(x)
```

is equivalent to

```
ops.relu6(x)
```

Refer to *mindspore.ops.relu6()* for more details.

mindspore.ops.SeLU

class mindspore.ops.**SeLU**

Activation function SeLU (Scaled exponential Linear Unit).

The activation function is defined as:

$$E_i = scale * \begin{cases} x_i, & \text{if } x_i \geq 0; \\ \text{alpha} * (\exp(x_i) - 1), & \text{otherwise.} \end{cases}$$

where $alpha$ and $scale$ are pre-defined constants($alpha = 1.67326324$ and $scale = 1.05070098$).

See more details in Self-Normalizing Neural Networks.

Inputs:

- **input_x** (Tensor) - Tensor of any dimension. The data type is int8, int32, float16, float32, float64(only CPU, GPU).

Outputs:

Tensor, with the same type and shape as the *input_x*.

Raises

TypeError – If dtype of *input_x* is not int8, int32, float16, float32, float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-1.0, 4.0, -8.0], [2.0, -5.0, 9.0]]), mindspore.float32)
>>> selu = ops.SeLU()
>>> output = selu(input_x)
>>> print(output)
[[-1.1113307 4.202804 -1.7575096]
 [ 2.101402 -1.7462534 9.456309 ]]
```

mindspore.ops.Sigmoid

class mindspore.ops.**Sigmoid**

```
prim = ops.Sigmoid()
out = prim(input)
```

is equivalent to

```
ops.sigmoid(input)
```

Refer to *mindspore.ops.sigmoid()* for more details.

mindspore.ops.Softmax

class mindspore.ops.**Softmax**(*axis=-1*)

Applies the Softmax operation to the input tensor on the specified axis.

Refer to *mindspore.ops.softmax()* for more details.

Parameters

axis (*Union[int, tuple], optional*) – The axis to perform the Softmax operation. Default: `-1` .

Inputs:

- **input** (Tensor) - Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions.

Outputs:

Tensor, with the same type and shape as the input.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> softmax = ops.Softmax()
>>> output = softmax(input)
>>> print(output)
[0.01165623 0.03168492 0.08612854 0.23412167 0.6364086 ]
```

mindspore.ops.Softplus

class mindspore.ops.**Softplus**

Softplus activation function.

Softplus is a smooth approximation to the ReLU function. It can be used to constrain the output of a machine to always be positive. The function is shown as follows:

$$\text{output} = \log(1 + \exp(x))$$

Inputs:

- **input_x** (Tensor) - Tensor of any dimension. Supported dtypes:
  - GPU/CPU: float16, float32, float64.
  - Ascend: float16, float32.

Outputs:

Tensor, with the same type and shape as the *input_x*.

Raises

- **TypeError** – If *input_x* is not a Tensor.

- **TypeError** – If the dtype of *input_x* is not float16, float32 or float64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([1, 2, 3, 4, 5]), mindspore.float32)
>>> softplus = ops.Softplus()
>>> output = softplus(input_x)
>>> print(output)
[1.3132615 2.126928  3.0485873 4.01815   5.0067153]
```

mindspore.ops.SoftShrink

class mindspore.ops.**SoftShrink**(*lambd=0.5*)

```
prim = ops.SoftShrink(lambd)
out = prim(input)
```

is equivalent to

```
ops.softshrink(input, lambd)
```

Refer to *mindspore.ops.softshrink()* for more details.

mindspore.ops.Softsign

class mindspore.ops.**Softsign**

Softsign activation function.

Refer to *mindspore.ops.softsign()* for more details.

Inputs:

- **input_x** (Tensor) - Tensor of shape $(N, *)$, where $*$ means, any number of additional dimensions, with float16 or float32 data type.

Outputs:

Tensor, with the same type and shape as the *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([0, -1, 2, 30, -30]), mindspore.float32)
>>> softsign = ops.Softsign()
>>> output = softsign(input_x)
>>> print(output)
[ 0.        -0.5         0.6666667  0.9677419 -0.9677419]
```

mindspore.ops.Tanh

class mindspore.ops.**Tanh**

```
prim = ops.Tanh()
out = prim(input)
```

is equivalent to

```
ops.tanh(input)
```

Refer to *mindspore.ops.tanh()* for more details.

18.2.4 Optimizer

API Name	Description	Supported Platforms
mindspore.ops.Adam	Updates gradients by the Adaptive Moment Estimation (Adam) algorithm.	Ascend GPU CPU
mindspore.ops.AdamWeightDecay	Updates gradients by the Adaptive Moment Estimation algorithm with weight decay (AdamWeightDecay).	Ascend GPU CPU
mindspore.ops.AdaptiveAvgPool2D	AdaptiveAvgPool2D operation.	Ascend GPU CPU
mindspore.ops.AdaptiveAvgPool3D	AdaptiveAvgPool3D operation.	Ascend GPU CPU
mindspore.ops.ApplyAdadelta	Updates relevant entries according to the adadelta scheme.	Ascend GPU CPU
mindspore.ops.ApplyAdagrad	Updates relevant entries according to the adagrad scheme.	Ascend GPU CPU
mindspore.ops.ApplyAdagradDA	Update *var* according to the proximal adagrad scheme.	Ascend GPU CPU
mindspore.ops.ApplyAdagradV2	Updates relevant entries according to the adagradv2 scheme.	Ascend GPU CPU
mindspore.ops.ApplyAdaMax	Updates relevant entries according to the adamax scheme.	Ascend GPU CPU
mindspore.ops.ApplyAdamWithAmsgradV2	Update var according to the Adam algorithm.	Ascend GPU CPU
mindspore.ops.ApplyAddSign	Updates relevant entries according to the AddSign algorithm.	Ascend GPU CPU
mindspore.ops.ApplyCenteredRMSProp	Optimizer that implements the centered RMSProp algorithm.	Ascend GPU CPU

continues on next page

Table 5 – continued from previous page

mindspore.ops.ApplyFtrl	Updates relevant entries according to the FTRL scheme.	Ascend GPU CPU
mindspore.ops.ApplyGradientDescent	Updates *var* by subtracting *alpha* * *delta* from it.	Ascend GPU CPU
mindspore.ops.ApplyMomentum	Optimizer that implements the Momentum algorithm.	Ascend GPU CPU
mindspore.ops.ApplyPowerSign	Updates relevant entries according to the AddSign algorithm.	Ascend GPU CPU
mindspore.ops.ApplyProximalAdagrad	Updates relevant entries according to the proximal adagrad algorithm.	Ascend GPU CPU
mindspore.ops.ApplyProximalGradientDescent	Updates relevant entries according to the FOBOS(Forward Backward Splitting) algorithm.	Ascend GPU CPU
mindspore.ops.ApplyRMSProp	Optimizer that implements the Root Mean Square prop(RMSProp) algorithm.	Ascend GPU CPU
mindspore.ops.LARSUpdate	Conducts LARS (layer-wise adaptive rate scaling) update on the sum of squares of gradient.	Ascend
mindspore.ops.SparseApplyAdagradV2	Updates relevant entries according to the adagrad scheme, one more epsilon attribute than SparseApplyAdagrad.	Ascend GPU CPU
mindspore.ops.SparseApplyProximalAdagrad	Updates relevant entries according to the proximal adagrad algorithm.	Ascend GPU
mindspore.ops.SGD	Computes the stochastic gradient descent.	Ascend GPU CPU
mindspore.ops.SparseApplyFtrl	Updates relevant entries according to the FTRL-proximal scheme For more details, please refer to *mindspore.nn.FTRL*.	Ascend GPU CPU
mindspore.ops.SparseApplyFtrlV2	The SparseApplyFtrlV2 interface is deprecated, please use the *mindspore.ops.SparseApplyFtrl* instead.	Deprecated

mindspore.ops.Adam

class mindspore.ops.**Adam**(*use_locking=False*, *use_nesterov=False*)

Updates gradients by the Adaptive Moment Estimation (Adam) algorithm.

The Adam algorithm is proposed in Adam: A Method for Stochastic Optimization.

For more details, please refer to *mindspore.nn.Adam*.

The updating formulas are as follows,

$$
\begin{aligned}
m &= \beta_1 * m + (1 - \beta_1) * g \\
v &= \beta_2 * v + (1 - \beta_2) * g * g \\
l &= \alpha * \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \\
w &= w - l * \frac{m}{\sqrt{v}+\epsilon}
\end{aligned}
$$

m represents the 1st moment vector, v represents the 2nd moment vector, g represents *gradient*, l represents scaling factor lr, β_1, β_2 represent *beta1* and *beta2*, t represents updating step while $beta_1^t(\beta_1^t)$ and $beta_2^t(\beta_2^t)$ represent *beta1_power* and *beta2_power*, α represents *learning_rate*, w represents *var*, ϵ represents *epsilon*.

Inputs of *var*, *m*, *v* and *gradient* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

- **use_locking** (*bool*) –Whether to enable a lock to protect variable tensors from being updated. If `True` , updates of the var, m, and v tensors will be protected by a lock. If `False` , the result is unpredictable. Default: `False` .
- **use_nesterov** (*bool*) –Whether to use Nesterov Accelerated Gradient (NAG) algorithm to update the gradients. If `True` , update the gradients using NAG. If `False` , update the gradients without using NAG. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Weights to be updated. The shape is $(N, *)$ where $*$ means, any number of additional dimensions. The data type can be float16 or float32.
- **m** (Union[Parameter, Tensor]) - The 1st moment vector in the updating formula, the shape should be the same as *var*.
- **v** (Union[Parameter, Tensor]) - the 2nd moment vector in the updating formula, the shape should be the same as *var*.
- **beta1_power** (float) - $beta_1^t(\beta_1^t)$ in the updating formula.
- **beta2_power** (float) - $beta_2^t(\beta_2^t)$ in the updating formula.
- **lr** (float) - l in the updating formula. The paper suggested value is 10^{-8}.
- **beta1** (float) - The exponential decay rate for the 1st moment estimations. The paper suggested value is 0.9.
- **beta2** (float) - The exponential decay rate for the 2nd moment estimations. The paper suggested value is 0.999.
- **epsilon** (float) - Term added to the denominator to improve numerical stability.
- **gradient** (Tensor) - Gradient, has the same shape and data type as *var*.

Outputs:

Tuple of 3 Tensor, the updated parameters.

- **var** (Tensor) - The same shape and data type as Inputs *var*.
- **m** (Tensor) - The same shape and data type as Inputs *m*.
- **v** (Tensor) - The same shape and data type as Inputs *v*.

Raises

- **TypeError** –If neither *use_locking* nor *use_nesterov* is a bool.
- **TypeError** –If *var*, *m* or *v* is not a Parameter.
- **TypeError** –If *beta1_power*, *beta2_power1*, *lr*, *beta1*, *beta2*, *epsilon* or *gradient* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> from mindspore import Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_adam = ops.Adam()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...         self.m = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="m")
```

(continues on next page)

(continued from previous page)

```
...         self.v = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="v")
...     def construct(self, beta1_power, beta2_power, lr, beta1, beta2, epsilon, grad):
...         out = self.apply_adam(self.var, self.m, self.v, beta1_power, beta2_power, lr,␣
↪beta1, beta2,
...                               epsilon, grad)
...         return out
...
>>> net = Net()
>>> gradient = Tensor(np.ones([2, 2]).astype(np.float32))
>>> output = net(0.9, 0.999, 0.001, 0.9, 0.999, 1e-8, gradient)
>>> print(net.var.asnumpy())
[[0.9996838 0.9996838]
 [0.9996838 0.9996838]]
```

mindspore.ops.AdamWeightDecay

class mindspore.ops.**AdamWeightDecay**(*use_locking=False*)

Updates gradients by the Adaptive Moment Estimation algorithm with weight decay (AdamWeightDecay).

The Adam algorithm is proposed in Adam: A Method for Stochastic Optimization. The AdamWeightDecay variant was proposed in Decoupled Weight Decay Regularization.

The updating formulas are as follows,

$$
\begin{aligned}
m &= \beta_1 * m + (1 - \beta_1) * g \\
v &= \beta_2 * v + (1 - \beta_2) * g * g \\
update &= \frac{m}{\sqrt{v} + \epsilon} \\
update &= \begin{cases} update + weight\_decay * w & \text{if } weight\_decay > 0 \\ update & \text{otherwise} \end{cases} \\
w &= w - lr * update
\end{aligned}
$$

m represents the 1st moment vector, v represents the 2nd moment vector, g represents *gradient*, β_1, β_2 represent *beta1* and *beta2*, lr represents *learning_rate*, w represents *var*, $decay$ represents *weight_decay*, ϵ represents *epsilon*.

Parameters

use_locking (*bool*) – Whether to enable a lock to protect variable tensors from being updated. If `True` , updates of the var, m, and v tensors will be protected by a lock. If `False` , the result is unpredictable. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Weights to be updated. The shape is $(N, *)$ where $*$ means any number of additional dimensions. The data type can be float16 or float32.
- **m** (Union[Parameter, Tensor]) - The 1st moment vector in the updating formula, it should have the the shape as *var*. The data type can be float16 or float32.
- **v** (Union[Parameter, Tensor]) - The 2nd moment vector in the updating formula, it should have the same shape as m.
- **lr** (float) - lr in the updating formula. The paper suggested value is 10^{-8}, the data type should be float32.
- **beta1** (float) - The exponential decay rate for the 1st moment estimations, the data type should be float32. The paper suggested value is 0.9
- **beta2** (float) - The exponential decay rate for the 2nd moment estimations, the data type should be float32. The paper suggested value is 0.999

- **epsilon** (float) - Term added to the denominator to improve numerical stability, the data type should be float32.
- **decay** (float) - The weight decay value, must be a scalar tensor with float32 data type. Default: `0.0`.
- **gradient** (Tensor) - Gradient, has the same shape as *var*.

Outputs:

Tuple of 3 Tensor, the updated parameters.

- **var** (Tensor) - The same shape and data type as *var*.
- **m** (Tensor) - The same shape and data type as *m*.
- **v** (Tensor) - The same shape and data type as *v*.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *lr*, *beta1*, *beta2*, *epsilon* or *decay* is not a float32.
- **TypeError** – If *var*, *m* or *v* is neither float16 nor float32.
- **TypeError** – If *gradient* is not a Tensor.
- **ValueError** – If *epsilon* <= 0.
- **ValueError** – If *beta1*, *beta2* is not in range (0.0,1.0).
- **ValueError** – If *decay* < 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import Tensor, Parameter, ops
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.adam_weight_decay = ops.AdamWeightDecay()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...         self.m = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="m")
...         self.v = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="v")
...     def construct(self, lr, beta1, beta2, epsilon, decay, grad):
...         out = self.adam_weight_decay(self.var, self.m, self.v, lr, beta1, beta2,
...                                      epsilon, decay, grad)
...         return out
>>> net = Net()
>>> gradient = Tensor(np.ones([2, 2]).astype(np.float32))
>>> output = net(0.001, 0.9, 0.999, 1e-8, 0.0, gradient)
>>> print(net.var.asnumpy())
[[0.999 0.999]
[0.999 0.999]]
```

mindspore.ops.AdaptiveAvgPool2D

class mindspore.ops.**AdaptiveAvgPool2D**(*output_size*)

AdaptiveAvgPool2D operation.

Refer to *mindspore.ops.adaptive_avg_pool2d()* for more details.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

output_size (*Union[int, tuple]*) –The target output size. *output_size* can be a tuple (H, W), or an int H for (H, H). H and W can be int or None. If it is None, it means the output size is the same as the input size.

Inputs:

- **input_x** (Tensor) - The input of AdaptiveAvgPool2D, which is a 3D or 4D tensor, with float16 ,float32 or float64 data type.

Outputs:

Tensor, with the same type as the *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: output_size=(None, 2)
>>> input_x = Tensor(np.array([[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                            [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]],
...                            [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]]),␣
↪mindspore.float32)
>>> adaptive_avg_pool_2d = ops.AdaptiveAvgPool2D((None, 2))
>>> output = adaptive_avg_pool_2d(input_x)
>>> print(output)
[[[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]
 [[1.5 2.5]
  [4.5 5.5]
  [7.5 8.5]]]
>>> # case 2: output_size=2
>>> adaptive_avg_pool_2d = ops.AdaptiveAvgPool2D(2)
>>> output = adaptive_avg_pool_2d(input_x)
>>> print(output)
[[[3. 4.]
  [6. 7.]]
 [[3. 4.]
```

(continues on next page)

(continued from previous page)

```
  [6. 7.]]
 [[3. 4.]
  [6. 7.]]]
>>> # case 3: output_size=(1, 2)
>>> adaptive_avg_pool_2d = ops.AdaptiveAvgPool2D((1, 2))
>>> output = adaptive_avg_pool_2d(input_x)
>>> print(output)
[[[4.5 5.5]]
 [[4.5 5.5]]
 [[4.5 5.5]]]
```

mindspore.ops.AdaptiveAvgPool3D

class mindspore.ops.**AdaptiveAvgPool3D**(*output_size*)

AdaptiveAvgPool3D operation.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.adaptive_avg_pool3d()* for more details.

Parameters

output_size (*Union[int, tuple]*) – Specify the size of output tensor. It can be a single int or a tuple of three ints.

Inputs:

- **x** (Tensor) - The input of AdaptiveAvgPool3D, which is a 5D or 4D tensor.

Outputs:

Tensor, with the same type as the *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import nn, Tensor
>>> from mindspore.ops import AdaptiveAvgPool3D
>>> class AdaptiveAvgPool3DNet(nn.Cell):
...     def __init__(self, output_size):
...         super(AdaptiveAvgPool3DNet, self).__init__()
...         self.output_size_ = output_size
...         self.adaptive_avg_pool_3d = AdaptiveAvgPool3D(self.output_size_)
...     def construct(self, x_):
...         return self.adaptive_avg_pool_3d(x_)
...
>>> output_size=(1,1,1)
>>> input_x_val = np.zeros((1,1,2,2,2))
>>> input_x_val[:,:,0,:,:]  += 1
>>> input_x = Tensor(input_x_val, mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> adaptive_avg_pool_3d = AdaptiveAvgPool3DNet(output_size)
>>> output = adaptive_avg_pool_3d(input_x)
>>> print(output)
[[[[[0.5]]]]]
```

mindspore.ops.ApplyAdadelta

class mindspore.ops.**ApplyAdadelta**

Updates relevant entries according to the adadelta scheme.

The Adadelta algorithm is proposed in ADADELTA: AN ADAPTIVE LEARNING RATE METHOD.

$$
\begin{aligned}
\text{accum} &= \rho * \text{accum} + (1 - \rho) * \text{grad}^2 \\
\text{update} &= \sqrt{\text{accum\_update} + \epsilon} * \frac{\text{grad}}{\sqrt{\text{accum} + \epsilon}} \\
\text{accum\_update} &= \rho * \text{accum\_update} + (1 - \rho) * \text{update}^2 \\
\text{var} &= \text{var} - \text{lr} * \text{update}
\end{aligned}
$$

where ρ represents *rho*, ϵ represents *epsilon*.

Inputs of *var*, *accum*, *accum_update* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Inputs:

- **var** (Union[Parameter, Tensor]) - Weights to be updated. With float32 or float16 data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - Accumulation to be updated, has the same shape and data type as *var*.
- **accum_update** (Union[Parameter, Tensor]) - Accum_update to be updated, has the same shape and data type as *var*.
- **lr** (Union[Number, Tensor]) - Learning rate, must be a scalar. With float32 or float16 data type.
- **rho** (Union[Number, Tensor]) - Decay rate, must be a scalar. With float32 or float16 data type.
- **epsilon** (Union[Number, Tensor]) - A small value added for numerical stability, must be a scalar. With float32 or float16 data type.
- **grad** (Tensor) - Gradients, has the same shape and data type as *var*.

Outputs:

Tuple of 3 Tensor, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.
- **accum_update** (Tensor) - The same shape and data type as *accum_update*.

Raises

- **TypeError** – If dtype of *var*, *accum*, *accum_update*, *lr*, *rho*, *epsilon* or *grad* is neither float16 nor float32.
- **TypeError** – If *accum_update*, *lr*, *rho* or *epsilon* is neither a Number nor a Tensor.
- **TypeError** – If the data type of *var*, *accum*, *accum_update* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import nn, Tensor, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_adadelta = ops.ApplyAdadelta()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
 ↪")
...         self.accum = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                 [0.2, 0.6]]).astype(np.float32)), name=
 ↪"accum")
...         self.accum_update = Parameter(Tensor(np.array([[0.9, 0.1],
...                                                        [0.7, 0.8]]).astype(np.float32)),
...                                       name="accum_update")
...     def construct(self, lr, rho, epsilon, grad):
...         out = self.apply_adadelta(self.var, self.accum, self.accum_update, lr, rho,␣
 ↪epsilon, grad)
...         return out
...
>>> net = Net()
>>> lr = Tensor(0.001, mindspore.float32)
>>> rho = Tensor(0.0, mindspore.float32)
>>> epsilon = Tensor(1e-6, mindspore.float32)
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(lr, rho, epsilon, grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.99051356e-01,  3.99683774e-01],
 [ 9.91633832e-02,  4.99105573e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 9.00000036e-02,  4.89999980e-01],
 [ 1.00000007e-02,  6.40000045e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 8.99990857e-01,  1.00000791e-01],
 [ 6.99930906e-01,  7.99999774e-01]]))
```

mindspore.ops.ApplyAdagrad

class mindspore.ops.**ApplyAdagrad**(*update_slots=True*)

Updates relevant entries according to the adagrad scheme. The Adagrad algorithm was proposed in Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. This module can adaptively assign different learning rates for each parameter in view of the uneven number of samples for different parameters.

$$
\begin{aligned}
accum &+= grad * grad \\
var &-= lr * grad * \frac{1}{\sqrt{accum}}
\end{aligned}
$$

Inputs of *var*, *accum* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

update_slots (*bool*) – If `True` , *accum* will be updated. Default: `True` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. With float or complex data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - Accumulation to be updated. The shape must be the same as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, must be a scalar. With float or complex data type.
- **grad** (Tensor) - A tensor for gradient. The shape must be the same as *var*.

Outputs:

Tuple of 2 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.

Raises

- **TypeError** –If dtype of *var*, *accum*, *lr* or *grad* is neither float nor complex.
- **TypeError** –If *lr* is neither a Number nor a Tensor.
- **TypeError** –If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_adagrad = ops.ApplyAdagrad()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
↪")
...         self.accum = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                 [0.2, 0.6]]).astype(np.float32)), name=
↪"accum")
...     def construct(self, lr, grad):
...         out = self.apply_adagrad(self.var, self.accum, lr, grad)
...         return out
...
>>> net = Net()
>>> lr = Tensor(0.001, mindspore.float32)
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(lr, grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.99638879e-01,  3.99296492e-01],
 [ 9.97817814e-02,  4.99281585e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 6.90000057e-01,  9.90000010e-01],
 [ 2.10000008e-01,  1.24000001e+00]]))
```

mindspore.ops.ApplyAdagradDA

class mindspore.ops.**ApplyAdagradDA**(*use_locking=False*)

Update *var* according to the proximal adagrad scheme. The Adagrad algorithm was proposed in Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

$$
\begin{aligned}
&grad\_accum+ = grad\\
&grad\_squared\_accum+ = grad * grad\\
&tmp\_val = \begin{cases} sign(grad\_accum) * max\left\{|grad\_accum| - l1 * global\_step, 0\right\} & \text{if } l1 > 0\\ grad\_accum & \text{otherwise} \end{cases}\\
&x\_value = -1 * lr * tmp\_val\\
&y\_value = l2 * global\_step * lr + \sqrt{grad\_squared\_accum}\\
&var = \frac{x\_value}{y\_value}
\end{aligned}
$$

Inputs of *var*, *gradient_accumulator*, *gradient_squared_accumulator* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool*) – If `True` , updating of the *var* and *accum* tensors will be protected by a lock. Otherwise the behavior is undefined, but may exhibit less contention. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **gradient_accumulator** (Union[Parameter, Tensor]) - The dict of mutable tensor $grad\_accum$. Must have the same shape as *var*.
- **gradient_squared_accumulator** (Union[Parameter, Tensor]) - The dict of mutable tensor $grad\_squared\_accum$. Must have the same shape as *var*.
- **grad** (Tensor) - A tensor for gradient. Must have the same shape as *var*.
- **lr** ([Number, Tensor]) - Scaling factor. Must be a scalar. With float32 or float16 data type.
- **l1** ([Number, Tensor]) - L1 regularization. Must be a scalar. With float32 or float16 data type.
- **l2** ([Number, Tensor]) - L2 regularization. Must be a scalar. With float32 or float16 data type.
- **global_step** ([Number, Tensor]) - Training step number. Must be a scalar. With int32 or int64 data type.

Outputs:

Tuple of 1 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.

Raises

- **TypeError** – If *var*, *gradient_accumulator* or *gradient_squared_accumulator* neither a Parameter nor a Tensor.
- **TypeError** – If *grad* is not a Tensor.
- **TypeError** – If *lr*, *l1*, *l2* or *global_step* is neither a Number nor a Tensor.
- **TypeError** – If use_locking is not a bool.
- **TypeError** – If dtype of *var*, *gradient_accumulator*, *gradient_squared_accumulator*, *grad*, *lr*, *l1* or *l2* is neither float16 nor float32.

- **TypeError** –If dtype of *gradient_accumulator*, *gradient_squared_accumulator* or *grad* is not same as *var*.
- **TypeError** –If dtype of *global_step* is not int32 nor int64.
- **ValueError** –If the shape size of *lr*, *l1*, *l2* and *global_step* is not 0.
- **TypeError** –If the data type of *var*, *gradient_accumulator*, *gradient_squared_accumulator* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class ApplyAdagradDANet(nn.Cell):
...     def __init__(self, use_locking=False):
...         super(ApplyAdagradDANet, self).__init__()
...         self.apply_adagrad_d_a = ops.ApplyAdagradDA(use_locking)
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4], [0.1, 0.5]]).astype(np.
↪float32)), name="var")
...         self.gradient_accumulator = Parameter(Tensor(np.array([[0.1, 0.3],
...                                                                [0.1, 0.5]]).astype(np.
↪float32)),
...                                               name="gradient_accumulator")
...         self.gradient_squared_accumulator = Parameter(Tensor(np.array([[0.2, 0.1],
...                                                                        [0.1, 0.2]]).
↪astype(np.float32)),
...                                                       name="gradient_squared_accumulator
↪")
...         self.gradient_accumulator = Parameter(Tensor(np.array([[0.1, 0.3],
...                                                                [0.1, 0.5]]).astype(np.
↪float32)),
...                                               name="gradient_accumulator")
...     def construct(self, grad, lr, l1, l2, global_step):
...         out = self.apply_adagrad_d_a(self.var, self.gradient_accumulator,
...                                      self.gradient_squared_accumulator, grad, lr, l1, l2,
↪ global_step)
...         return out
...
>>> net = ApplyAdagradDANet()
>>> grad = Tensor(np.array([[0.3, 0.4], [0.1, 0.2]]).astype(np.float32))
>>> lr = Tensor(0.001, mstype.float32)
>>> l1 = Tensor(0.001, mstype.float32)
>>> l2 = Tensor(0.001, mstype.float32)
>>> global_step = Tensor(2, mstype.int32)
>>> output = net(grad, lr, l1, l2, global_step)
>>> print(output)
[[-0.00073906, -0.00136889],
 [-0.00059699, -0.00142478]]
```

mindspore.ops.ApplyAdagradV2

class mindspore.ops.**ApplyAdagradV2**(*epsilon*, *update_slots=True*)

Updates relevant entries according to the adagradv2 scheme.

$$accum += grad * grad$$
$$var -= lr * grad * \frac{1}{\sqrt{accum} + \epsilon}$$

where ϵ represents *epsilon*.

Inputs of *var*, *accum* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Note: The difference is that *ApplyAdagradV2* has one more small constant value ϵ than *ApplyAdagrad*.

Parameters

- **epsilon** (*float*) – A small value added for numerical stability.
- **update_slots** (*bool*) – If `True` , *accum* will be updated. Default: `True` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. With float16 or float32 data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - Accumulation to be updated. The shape must be the same as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, must be a float number or a scalar tensor with float16 or float32 data type.
- **grad** (Tensor) - A tensor for gradient. The shape must be the same as *var*.

Outputs:

Tuple of 2 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.

Raises

- **TypeError** – If dtype of *var*, *accum*, *lr* or *grad* is neither float16 nor float32.
- **TypeError** – If *lr* is neither a Number nor a Tensor.
- **TypeError** – If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_adagrad_v2 = ops.ApplyAdagradV2(epsilon=1e-6)
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
 ↪")
...         self.accum = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                 [0.2, 0.6]]).astype(np.float32)), name=
 ↪"accum")
...     def construct(self, lr, grad):
...         out = self.apply_adagrad_v2(self.var, self.accum, lr, grad)
...         return out
...
>>> net = Net()
>>> lr = Tensor(0.001, mindspore.float32)
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(lr, grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.99638879e-01,  3.99296492e-01],
 [ 9.97817814e-02,  4.99281585e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 6.90000057e-01,  9.90000010e-01],
 [ 2.10000008e-01,  1.24000001e+00]]))
```

mindspore.ops.ApplyAdaMax

class mindspore.ops.**ApplyAdaMax**

Updates relevant entries according to the adamax scheme.

The updating formulas are as follows,

$$
\begin{aligned}
m_{t+1} &= \beta_1 * m_t + (1 - \beta_1) * g \\
v_{t+1} &= \max(\beta_2 * v_t, |g|) \\
var &= var - \frac{l}{1 - \beta_1^{t+1}} * \frac{m_{t+1}}{v_{t+1} + \epsilon}
\end{aligned}
$$

t represents updating step while m represents the 1st moment vector, m_t is the last moment of m_{t+1}, v represents the 2nd moment vector, v_t is the last moment of v_{t+1}, l represents scaling factor lr, g represents $grad$, β_1, β_2 represent $beta1$ and $beta2$, β_1^{t+1} represents $beta1\_power$, var represents the variable to be updated, ϵ represents $epsilon$.

Inputs of var, m, v and $grad$ comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. With float32 or float16 data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **m** (Union[Parameter, Tensor]) - The 1st moment vector in the updating formula, has the same shape as var. With float32 or float16 data type.

- **v** (Union[Parameter, Tensor]) - The 2nd moment vector in the updating formula. Mean square gradients with the same shape as *var*. With float32 or float16 data type.
- **beta1_power** (Union[Number, Tensor]) - $beta_1^t$ in the updating formula, must be a scalar. With float32 or float16 data type.
- **lr** (Union[Number, Tensor]) - Learning rate, l in the updating formula, must be a scalar. With float32 or float16 data type.
- **beta1** (Union[Number, Tensor]) - The exponential decay rate for the 1st moment estimations, must be a scalar. With float32 or float16 data type.
- **beta2** (Union[Number, Tensor]) - The exponential decay rate for the 2nd moment estimations, must be a scalar. With float32 or float16 data type.
- **epsilon** (Union[Number, Tensor]) - A small value added for numerical stability, must be a scalar. With float32 or float16 data type.
- **grad** (Tensor) - A tensor for gradient, has the same shape as *var*. With float32 or float16 data type.

Outputs:

Tuple of 3 Tensor, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **m** (Tensor) - The same shape and data type as *m*.
- **v** (Tensor) - The same shape and data type as *v*.

Raises

- **TypeError** –If dtype of *var*, *m*, *v*, *beta_power*, *lr*, *beta1*, *beta2*, *epsilon* or *grad* is neither float16 nor float32.
- **TypeError** –If *beta_power*, *lr*, *beta1*, *beta2* or *epsilon* is neither a Number nor a Tensor.
- **TypeError** –If *grad* is not a Tensor.
- **TypeError** –If the data type of *var*, *m*, *v* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_ada_max = ops.ApplyAdaMax()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
↪")
...         self.m = Parameter(Tensor(np.array([[0.6, 0.5],
...                                             [0.2, 0.6]]).astype(np.float32)), name="m")
...         self.v = Parameter(Tensor(np.array([[0.9, 0.1],
...                                             [0.7, 0.8]]).astype(np.float32)), name="v")
...     def construct(self, beta1_power, lr, beta1, beta2, epsilon, grad):
```

(continues on next page)

(continued from previous page)

```
...         out = self.apply_ada_max(self.var, self.m, self.v, beta1_power, lr, beta1, beta2,
 ↪ epsilon, grad)
...         return out
...
>>> net = Net()
>>> beta1_power =Tensor(0.9, mindspore.float32)
>>> lr = Tensor(0.001, mindspore.float32)
>>> beta1 = Tensor(0.9, mindspore.float32)
>>> beta2 = Tensor(0.99, mindspore.float32)
>>> epsilon = Tensor(1e-10, mindspore.float32)
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(beta1_power, lr, beta1, beta2, epsilon, grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.93602717e-01,  3.92571449e-01],
 [ 9.72582996e-02,  4.92249995e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.69999993e-01,  5.19999981e-01],
 [ 1.89999998e-01,  6.20000005e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 8.90999973e-01,  6.99999988e-01],
 [ 6.93000019e-01,  8.00000012e-01]]))
```

mindspore.ops.ApplyAdamWithAmsgradV2

class mindspore.ops.**ApplyAdamWithAmsgradV2**(*use_locking=False*)

Update var according to the Adam algorithm.

$$
\begin{aligned}
lr_t &:= learning\_rate * \sqrt{1 - \beta_2^t} / (1 - \beta_1^t) \\
m_t &:= \beta_1 * m_{t-1} + (1 - \beta_1) * g \\
v_t &:= \beta_2 * v_{t-1} + (1 - \beta_2) * g * g \\
\hat{v}_t &:= \max(\hat{v}_{t-1}, v_t) \\
var &:= var - lr_t * m_t / (\sqrt{\hat{v}_t} + \epsilon)
\end{aligned}
$$

t represents updating step while m represents the 1st moment vector, v represents the 2nd moment vector, $\hat{v}_t$ represents *vhat*, lr represents learning rate, g represents *grad*, β_1, β_2 represent *beta1* and *beta2*, β_1^t represents *beta1_power*, β_2^t represents *beta2_power*, var represents the variable to be updated, ϵ represents *epsilon*.

All of the inputs are consistent with implicit type conversion rules, which ensure that the data types are the same. If they have different data types, the lower precision data type will be converted to the data type with relatively higher precision.

Parameters

use_locking (*bool*) – If `True` , updating of the *var*, *m*, and *v* tensors will be protected by a lock; Otherwise some contention may occur. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. The data type can be float16, float32 or float64.
- **m** (Union[Parameter, Tensor]) - The 1st moment vector in the updating formula, the shape should be the same as *var*.
- **v** (Union[Parameter, Tensor]) - The 2nd moment vector in the updating formula, the shape should be the same as *var*.
- **vhat** (Union[Parameter, Tensor]) - $\hat{v}_t$ in the updating formula, the shape and data type value should be the same as *var*.
- **beta1_power** (Union[float, Tensor]) - $beta_1^t(\beta_1^t)$ in the updating formula, with float16, float32 or float64 data type.
- **beta2_power** (Union[float, Tensor]) - $beta_2^t(\beta_2^t)$ in the updating formula, with float16, float32 or float64 data type.

- **lr** (Union[float, Tensor]) - Learning rate, with float16, float32 or float64 data type.
- **beta1** (Union[float, Tensor]) - Exponential decay rate of the first moment. The data type can be float16, float32 or float64.
- **beta2** (Union[float, Tensor]) - Exponential decay rate of the second moment. The data type can be float16, float32 or float64.
- **epsilon** (Union[float, Tensor]) - A value added to the denominator to ensure numerical stability. The data type can be float16, float32 or float64.
- **grad** (Tensor) - The gradient, has the same shape as *var*.

Outputs:

Tuple of 4 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **m** (Tensor) - The same shape and data type as *m*.
- **v** (Tensor) - The same shape and data type as *v*.
- **vhat** (Tensor) - The same shape and data type as *vhat*.

Raises

- **TypeError** –If *var*, *m*, *v*, *vhat* neither a Parameter nor a Tensor.
- **TypeError** –If dtype of *var*, *m*, *v*, *vhat*, *beta1_power*, *beta2_power*, *lr*, *beta1* , *beta2* , *epsilon* or *grad* is not float64, float32 or float16.
- **RuntimeError** –If the data type of *var*, *m*, *v* , *vhat* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> import mindspore.nn as nn
>>> from mindspore import Tensor, Parameter
>>> import numpy as np
>>> class ApplyAdamWithAmsgradNet(nn.Cell):
...     def __init__(self, use_locking=False):
...         super(ApplyAdamWithAmsgradNet, self).__init__()
...         self.apply_adam_with_amsgrad = ops.ApplyAdamWithAmsgradV2(use_locking)
...         self.var = Parameter(Tensor(np.array([[0.2, 0.2], [0.2, 0.2]]).astype(np.
↪float32)), name="var")
...         self.m = Parameter(Tensor(np.array([[0.1, 0.2], [0.4, 0.3]]).astype(np.float32)),
↪ name="m")
...         self.v = Parameter(Tensor(np.array([[0.2, 0.1], [0.3, 0.4]]).astype(np.float32)),
↪ name="v")
...         self.vhat = Parameter(Tensor(np.array([[0.1, 0.2], [0.6, 0.2]]).astype(np.
↪float32)), name="vhat")
...         self.beta1 = 0.8
...         self.beta2 = 0.999
...         self.epsilon = 1e-8
...         self.beta1_power = 0.9
...         self.beta2_power = 0.999
```

(continues on next page)

(continued from previous page)

```
...         self.lr = 0.01
...
...     def construct(self, grad):
...         out = self.apply_adam_with_amsgrad(self.var, self.m, self.v, self.vhat,
...                                            self.beta1_power, self.beta2_power, self.lr,
...                                            self.beta1, self.beta2, self.epsilon, grad)
...         return out
>>> net = ApplyAdamWithAmsgradNet()
>>> grad = Tensor(np.array([[0.4, 0.2], [0.2, 0.3]]).astype(np.float32))
>>> output = net(grad)
>>> print(net.var.asnumpy())
[[0.19886853 0.1985858 ]
[0.19853032 0.19849943]]
```

mindspore.ops.ApplyAddSign

class mindspore.ops.**ApplyAddSign**

Updates relevant entries according to the AddSign algorithm.

$$
\begin{aligned}
m_{t+1} &= \beta * m_t + (1 - \beta) * g \\
\text{update} &= (\alpha + \text{sign\_decay} * sign(g) * sign(m)) * g \\
var &= var - lr_{t+1} * \text{update}
\end{aligned}
$$

t represents updating step while m represents the 1st moment vector, m_t is the last moment of m_{t+1}, lr represents scaling factor lr, g represents *grad*, α represents *alpha*, β represents *beta*.

The data type of all inputs must be float16 or float32 on Ascend and float16, float32 or float64 on CPU and GPU.

Inputs of *var*, *accum* and *grad* , *sign_decay* and *beta* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable tensor to be updated. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **m** (Union[Parameter, Tensor]) - Variable tensor to be updated, has the same data type as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, must be a scalar.
- **alpha** (Union[Number, Tensor]) - Must be a scalar.
- **sign_decay** (Union[Number, Tensor]) - Must be a scalar.
- **beta** (Union[Number, Tensor]) - The exponential decay rate, must be a scalar.
- **grad** (Tensor) - A tensor of the same shape as *var*, for the gradient.

Outputs:

Tuple of 2 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **m** (Tensor) - The same shape and data type as *m*.

Raises

- **TypeError** –If dtype of *var*, *lr* and *alpha* is not float16, float32 or float64.

- **TypeError** – If dtype of *sign_decay* and *beta* are both not float16, float32 or float64.
- **TypeError** – If *lr*, *alpha* or *sign_decay* is neither a Number nor a Tensor.
- **TypeError** – If *grad* is not a Tensor.
- **TypeError** – If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_add_sign = ops.ApplyAddSign()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
↪")
...         self.m = Parameter(Tensor(np.array([[0.6, 0.5],
...                                             [0.2, 0.6]]).astype(np.float32)), name="m")
...         self.lr = 0.001
...         self.alpha = 1.0
...         self.sign_decay = 0.99
...         self.beta = 0.9
...     def construct(self, grad):
...         out = self.apply_add_sign(self.var, self.m, self.lr, self.alpha, self.sign_decay,
↪ self.beta, grad)
...         return out
...
>>> net = Net()
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.99403024e-01,  3.98607016e-01],
 [ 9.98010039e-02,  4.98407990e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.70000052e-01,  5.19999981e-01],
 [ 1.89999998e-01,  6.20000064e-01]]))
```

mindspore.ops.ApplyCenteredRMSProp

class mindspore.ops.**ApplyCenteredRMSProp**(*use_locking=False*)

Optimizer that implements the centered RMSProp algorithm. Please refer to the usage in source code of *mindspore.nn.RMSProp*.

The updating formulas of ApplyCenteredRMSProp algorithm are as follows,

$$
\begin{aligned}
g_{t+1} &= \rho g_t + (1-\rho)\nabla Q_i(w) \\
s_{t+1} &= \rho s_t + (1-\rho)(\nabla Q_i(w))^2 \\
m_{t+1} &= \beta m_t + \frac{\eta}{\sqrt{s_{t+1} - g_{t+1}^2 + \epsilon}}\nabla Q_i(w) \\
w &= w - m_{t+1}
\end{aligned}
$$

where w represents *var*, which will be updated. g_{t+1} represents *mean_gradient*, g_t is the last moment of g_{t+1}. s_{t+1} represents *mean_square*, s_t is the last moment of s_{t+1}, m_{t+1} represents *moment*, m_t is the last moment of m_{t+1}. ρ represents *decay*. β is the momentum term, represents *momentum*. ϵ is a smoothing term to avoid division by zero, represents *epsilon*. η represents *learning_rate*. $\nabla Q_i(w)$ represents *grad*.

Note: The difference between *ApplyCenteredRMSProp* and *ApplyRMSProp* is that the former uses the centered RMSProp algorithm, and the centered RRMSProp algorithm uses an estimate of the centered second moment(i.e., the variance) for normalization, as opposed to regular RMSProp, which uses the (uncertained) second moment. This often helps with training, but is slightly more expensive in terms of computation and memory.

Warning: In dense implementation of this algorithm, *mean_gradient*, *mean_square*, and *moment* will update even if the *grad* is zero. But in this sparse implementation, *mean_gradient*, *mean_square*, and *moment* will not update in iterations during which the *grad* is zero.

Parameters

use_locking (*bool*) – Whether to enable a lock to protect the variable and accumulation tensors from being updated. Default: `False`.

Inputs:

- **var** (Parameter) - Weights to be updated.
- **mean_gradient** (Tensor) - Mean gradients, must be the same type as *var*.
- **mean_square** (Tensor) - Mean square gradients, must be the same type as *var*.
- **moment** (Tensor) - Delta of *var*, must be the same type as *var*.
- **grad** (Tensor) - Gradient, must be the same type as *var*.
- **learning_rate** (Union[Number, Tensor]) - Learning rate. Must be a float number or a scalar tensor with float16 or float32 data type.
- **decay** (float) - Decay rate.
- **momentum** (float) - Momentum.
- **epsilon** (float) - Ridge term.

Outputs:

Tensor, parameters to be updated.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *var*, *mean_gradient*, *mean_square*, *moment* or *grad* is not a Tensor.
- **TypeError** – If *learing_rate* is neither a Number nor a Tensor.
- **TypeError** – If dtype of *learing_rate* is neither float16 nor float32.
- **TypeError** – If *decay*, *momentum* or *epsilon* is not a float.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_centerd_rms_prop = ops.ApplyCenteredRMSProp()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...
...     def construct(self, mean_grad, mean_square, moment, grad, decay, momentum, epsilon,␣
↪lr):
...         out = self.apply_centerd_rms_prop(self.var, mean_grad, mean_square, moment, grad,
...                                           lr, decay, momentum, epsilon)
...         return out
...
>>> net = Net()
>>> mean_grad = Tensor(np.ones([2, 2]).astype(np.float32))
>>> mean_square = Tensor(np.ones([2, 2]).astype(np.float32))
>>> moment = Tensor(np.ones([2, 2]).astype(np.float32))
>>> grad = Tensor(np.ones([2, 2]).astype(np.float32))
>>> output = net(mean_grad, mean_square, moment, grad, 0.0, 1e-10, 0.001, 0.01)
>>> print(net.var.asnumpy())
[[0.68377227  0.68377227]
 [0.68377227  0.68377227]]
```

mindspore.ops.ApplyFtrl

class mindspore.ops.**ApplyFtrl**(*use_locking=False*)

Updates relevant entries according to the FTRL scheme.

For more details, please refer to *mindspore.nn.FTRL*.

Note:

- Currently, only positive numbers are supported on the Ascend platform, and the calculation results for other scenarios are not defined.
- Inputs of *var*, *accum*, *linear* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool*) – Use locks for updating operation if `True` . Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - The variable to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - The accumulation to be updated, must be same shape as *var*.
- **linear** (Union[Parameter, Tensor]) - The linear coefficient to be updated, must be same shape as *var*.
- **grad** (Tensor) - Gradient. The data type must be float16 or float32.

- **lr** (Union[Number, Tensor]) - The learning rate value, must be positive. Default: `0.001` . It must be a float number or a scalar tensor with float16 or float32 data type.
- **l1** (Union[Number, Tensor]) - l1 regularization strength, must be greater than or equal to zero. Default: `0.0` . It must be a float number or a scalar tensor with float16 or float32 data type.
- **l2** (Union[Number, Tensor]) - l2 regularization strength, must be greater than or equal to zero. Default: `0.0` . It must be a float number or a scalar tensor with float16 or float32 data type.
- **lr_power** (Union[Number, Tensor]) - Learning rate power controls how the learning rate decreases during training, must be less than or equal to zero. Use fixed learning rate if lr_power is zero. Default: `-0.5` . It must be a float number or a scalar tensor with float16 or float32 data type.

Outputs:

- **var** (Tensor) - Represents the updated *var*. As the input parameters or tensors has been updated in-place, this value is always zero when the platform is GPU.

Raises

- **TypeError** –If *use_locking* is not a bool.
- **TypeError** –If dtype of *var*, *grad*, *lr*, *l1*, *l2* or *lr_power* is neither float16 nor float32.
- **TypeError** –If *lr*, *l1*, *l2* or *lr_power* is neither a Number nor a Tensor.
- **TypeError** –If *grad* is not a Tensor.
- **TypeError** –If the parameter or tensor types of *var*, *accum* and *linear* are inconsistent.
- **TypeError** –If the parameter or tensor types of *grad*, *lr*, *l1*, *l2*, *lr_power* are inconsistent with *var* and the precision is greater than *var*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class ApplyFtrlNet(nn.Cell):
...     def __init__(self):
...         super(ApplyFtrlNet, self).__init__()
...         self.apply_ftrl = ops.ApplyFtrl()
...         self.lr = 0.001
...         self.l1 = 0.0
...         self.l2 = 0.0
...         self.lr_power = -0.5
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
↪")
...         self.accum = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                 [0.2, 0.6]]).astype(np.float32)), name=
↪"accum")
...         self.linear = Parameter(Tensor(np.array([[0.9, 0.1],
...                                                  [0.7, 0.8]]).astype(np.float32)), name=
↪"linear")
...
...     def construct(self, grad):
```

(continues on next page)

(continued from previous page)

```
...         out = self.apply_ftrl(self.var, self.accum, self.linear, grad, self.lr, self.l1,␣
↪self.l2,
...                               self.lr_power)
...         return out
...
>>> net = ApplyFtrlNet()
>>> input_x = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(input_x)
>>> print(net.var.asnumpy())
[[ 0.0390525  0.11492836]
 [ 0.00066425 0.15075898]]
```

mindspore.ops.ApplyGradientDescent

class mindspore.ops.**ApplyGradientDescent**

Updates *var* by subtracting *alpha* * *delta* from it.

$$var = var - \alpha * \delta$$

where α represents *alpha*, δ represents *delta*.

Inputs of *var* and *delta* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable tensor to be updated. With float32 or float16 data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **alpha** (Union[Number, Tensor]) - Scaling factor, must be a scalar. With float32 or float16 data type.
- **delta** (Tensor) - A tensor for the change, has the same shape as *var*.

Outputs:

Tensor, represents the updated *var*.

Raises

- **TypeError** – If dtype of *var* or *alpha* is neither float16 nor float32.
- **TypeError** – If *delta* is not a Tensor.
- **TypeError** – If *alpha* is neither a Number nor a Tensor.
- **TypeError** – If the data type of *var* and *delta* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_gradient_descent = ops.ApplyGradientDescent()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...         self.alpha = 0.001
...     def construct(self, delta):
...         out = self.apply_gradient_descent(self.var, self.alpha, delta)
...         return out
...
>>> net = Net()
>>> delta = Tensor(np.array([[0.1, 0.1], [0.1, 0.1]]).astype(np.float32))
>>> output = net(delta)
>>> print(output)
[[0.9999 0.9999]
 [0.9999 0.9999]]
```

mindspore.ops.ApplyMomentum

class mindspore.ops.**ApplyMomentum**(*use_nesterov=False*, *use_locking=False*, *gradient_scale=1.0*)

Optimizer that implements the Momentum algorithm.

Refer to the paper On the importance of initialization and momentum in deep learning for more details.

Inputs of *variable*, *accumulation* and *gradient* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Refer to *mindspore.nn.Momentum* for more details about the formula and usage.

Parameters

- **use_locking** (*bool, optional*) – Whether to enable a lock to protect the variable and accumulation tensors from being updated. Default: `False`.
- **use_nesterov** (*bool, optional*) – Enable Nesterov momentum. Default: `False`.
- **gradient_scale** (*float, optional*) – The scale of the gradient. Default: `1.0`.

Inputs:

- **variable** (Union[Parameter, Tensor]) - Weights to be updated. Data type must be float64, int64, float, float16, int16, int32, int8, uint16, uint32, uint64, uint8, complex64, complex128.
- **accumulation** (Union[Parameter, Tensor]) - Accumulated gradient value by moment weight, has the same data type with *variable*.
- **learning_rate** (Union[Number, Tensor]) - The learning rate value, must be a float64, int64, float, float16, int16, int32, int8, uint16, uint32, uint64, uint8, complex64, complex128 number or a scalar tensor with float64, int64, float, float16, int16, int32, int8, uint16, uint32, uint64, uint8, complex64, complex128 data type.
- **gradient** (Tensor) - Gradient, has the same data type as *variable*.
- **momentum** (Union[Number, Tensor]) - Momentum, must be a float64, int64, float, float16, int16, int32, int8, uint16, uint32, uint64, uint8, complex64, complex128 number or a scalar tensor with float64, int64, float, float16, int16, int32, int8, uint16, uint32, uint64, uint8, complex64, complex128 data type.

Outputs:

Tensor, parameters to be updated.

Raises

- **TypeError** – If the *use_locking* or *use_nesterov* is not a bool or *gradient_scale* is not a float.
- **TypeError** – If the data type of *var*, *accum* and *grad* conversion is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_momentum = ops.ApplyMomentum()
...         self.variable = Parameter(Tensor(np.array([[0.6, 0.4],
...                                                    [0.1, 0.5]]).astype(np.float32)), name=
 ↪"variable")
...         self.accumulate = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                    [0.2, 0.6]]).astype(np.float32)), name=
 ↪"accumulate")
...     def construct(self, lr, grad, moment):
...         out = self.apply_momentum(self.variable, self.accumulate, lr, grad, moment)
...         return out
>>> net = Net()
>>> lr = Tensor(0.1, mindspore.float32)
>>> moment = Tensor(0.9, mindspore.float32)
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(lr, grad, moment)
>>> print(output)
[[0.51600003 0.285     ]
[0.072      0.366     ]]
```

mindspore.ops.ApplyPowerSign

class mindspore.ops.**ApplyPowerSign**

Updates relevant entries according to the AddSign algorithm.

The AddSign algorithm was proposed in Neural Optimizer Search with Reinforcement Learning.

$$
\begin{aligned}
m_{t+1} &= \beta * m_t + (1 - \beta) * g \\
\text{update} &= \exp(\text{logbase} * \text{sign\_decay} * sign(g) * sign(m)) * g \\
var &= var - lr_{t+1} * \text{update}
\end{aligned}
$$

t represents updating step while m represents the 1st moment vector, m_t is the last moment of m_{t+1}, lr represents scaling factor lr, g represents *grad*, β represents *beta*.

All of inputs comply with the implicit type conversion rules to make the data types consistent. If *lr*, *logbase*, *sign_decay* or *beta* is a number, the number is automatically converted to Tensor, and the data type is consistent with the Tensor data type involved

in the operation. If inputs are tensors and have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Note: On Ascend, input data type of float64 is currently not supported.

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable tensor to be updated. With float64, float32 or float16 data type. If data type of *var* is float16, all inputs must have the same data type as *var*. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **m** (Union[Parameter, Tensor]) - Variable tensor to be updated, has the same shape as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, should be a scalar or Tensor with float64, float32 or float16 data type.
- **logbase** (Union[Number, Tensor]) - Should be a scalar or Tensor with float64, float32 or float16 data type.
- **sign_decay** (Union[Number, Tensor]) - Should be a scalar or Tensor with float64, float32 or float16 data type.
- **beta** (Union[Number, Tensor]) - The exponential decay rate, should be a scalar or Tensor with float64, float32 or float16 data type.
- **grad** (Tensor) - A tensor of the same shape as *var*, for the gradient.

Outputs:

Tuple of 2 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **m** (Tensor) - The same shape and data type as *m*.

Raises

- **TypeError** –If dtype of *var*, *lr*, *logbase*, *sign_decay*, *beta* or *grad* is not one of float16,
- **float32 or float64.** –
- **TypeError** –If *lr*, *logbase*, *sign_decay* or *beta* is neither a Number nor a Tensor.
- **TypeError** –If *grad* is not a Tensor.
- **TypeError** –If the data type of *lr*, *logbase*, *sign_decay* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_power_sign = ops.ApplyPowerSign()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
↪")
```

(continues on next page)

(continued from previous page)

```
...         self.m = Parameter(Tensor(np.array([[0.6, 0.5],
...                                             [0.2, 0.6]]).astype(np.float32)), name="m")
...         self.lr = 0.001
...         self.logbase = np.e
...         self.sign_decay = 0.99
...         self.beta = 0.9
...     def construct(self, grad):
...         out = self.apply_power_sign(self.var, self.m, self.lr, self.logbase,
...                                     self.sign_decay, self.beta, grad)
...         return out
...
>>> net = Net()
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.95575690e-01,  3.89676481e-01],
 [ 9.85252112e-02,  4.88201708e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.70000052e-01,  5.19999981e-01],
 [ 1.89999998e-01,  6.20000064e-01]]))
```

mindspore.ops.ApplyProximalAdagrad

class mindspore.ops.**ApplyProximalAdagrad**(*use_locking=False*)

Updates relevant entries according to the proximal adagrad algorithm. The proximal adagrad algorithm was proposed in Efficient Learning using Forward-Backward Splitting.

$$
\begin{aligned}
&accum += grad * grad \\
&\text{prox\_v} = var - lr * grad * \frac{1}{\sqrt{accum}} \\
&var = \frac{sign(\text{prox\_v})}{1 + lr * l2} * \max(|\text{prox\_v}| - lr * l1, 0)
\end{aligned}
$$

Inputs of *var*, *accum* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool, optional*) – If `True` , the var and accumulation tensors will be protected from being updated. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - Accumulation to be updated, must have the same shape and dtype as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, must be a scalar. The data type must be float16 or float32.
- **l1** (Union[Number, Tensor]) - l1 regularization strength, must be a scalar. The data type must be float16 or float32.
- **l2** (Union[Number, Tensor]) - l2 regularization strength, must be a scalar. The data type must be float16 or float32.
- **grad** (Tensor) - Gradient with the same shape and dtype as *var*.

Outputs:

Tuple of 2 Tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.

Raises

- **TypeError** – If *use_blocking* is not a bool.
- **TypeError** – If dtype of *var*, *lr*, *l1* or *l2* is neither float16 nor float32.
- **TypeError** – If *lr*, *l1* or *l2* is neither a Number nor a Tensor.
- **TypeError** – If *grad* is not a Tensor.
- **TypeError** – If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_proximal_adagrad = ops.ApplyProximalAdagrad()
...         self.var = Parameter(Tensor(np.array([[0.6, 0.4],
...                                               [0.1, 0.5]]).astype(np.float32)), name="var
 ↪")
...         self.accum = Parameter(Tensor(np.array([[0.6, 0.5],
...                                                 [0.2, 0.6]]).astype(np.float32)), name=
 ↪"accum")
...         self.lr = 0.01
...         self.l1 = 0.0
...         self.l2 = 0.0
...     def construct(self, grad):
...         out = self.apply_proximal_adagrad(self.var, self.accum, self.lr, self.l1, self.
 ↪l2, grad)
...         return out
...
>>> net = Net()
>>> grad = Tensor(np.array([[0.3, 0.7], [0.1, 0.8]]).astype(np.float32))
>>> output = net(grad)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 5.96388459e-01,  3.92964751e-01],
 [ 9.78178233e-02,  4.92815793e-01]]), Tensor(shape=[2, 2], dtype=Float32, value=
[[ 6.90000057e-01,  9.90000010e-01],
 [ 2.10000008e-01,  1.24000001e+00]]))
```

mindspore.ops.ApplyProximalGradientDescent

class mindspore.ops.**ApplyProximalGradientDescent**

Updates relevant entries according to the FOBOS(Forward Backward Splitting) algorithm. Refer to the paper Efficient Learning using Forward-Backward Splitting for more details.

$$\begin{aligned}\text{prox\_v} &= var - \alpha * \delta \\ var &= \frac{sign(\text{prox\_v})}{1+\alpha * l2} * \max(|\text{prox\_v}| - \alpha * l1, 0)\end{aligned}$$

where α represents *alpha*, δ represents *delta*.

Inputs of *var* and *delta* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable tensor to be updated. With float32 or float16 data type. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **alpha** (Union[Number, Tensor]) - Scaling factor, must be a scalar. With float32 or float16 data type.
- **l1** (Union[Number, Tensor]) - l1 regularization strength, must be a scalar. With float32 or float16 data type.
- **l2** (Union[Number, Tensor]) - l2 regularization strength, must be a scalar. With float32 or float16 data type.
- **delta** (Tensor) - A tensor for the change.

Outputs:

Tensor, represents the updated *var*.

Raises

- **TypeError** – If dtype of *var*, *alpha*, *l1* or *l2* is neither float16 nor float32.
- **TypeError** – If *alpha*, *l1* or *l2* is neither a Number nor a Tensor.
- **TypeError** – If *delta* is not a Tensor.
- **TypeError** – If the data type of *var*, and *delta* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_proximal_gradient_descent = ops.ApplyProximalGradientDescent()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...         self.alpha = 0.001
...         self.l1 = 0.1
...         self.l2 = 0.1
...     def construct(self, delta):
...         out = self.apply_proximal_gradient_descent(self.var, self.alpha, self.l1, self.
↪l2, delta)
```

(continues on next page)

(continued from previous page)

```
...         return out
...
>>> net = Net()
>>> delta = Tensor(np.array([[0.1, 0.1], [0.1, 0.1]]).astype(np.float32))
>>> output = net(delta)
>>> print(output)
[[0.99969995 0.99969995]
 [0.99969995 0.99969995]]
```

mindspore.ops.ApplyRMSProp

class mindspore.ops.**ApplyRMSProp**(*use_locking=False*)

Optimizer that implements the Root Mean Square prop(RMSProp) algorithm. Please refer to the usage in source code of *mindspore.nn.RMSProp*.

The updating formulas of ApplyRMSProp algorithm are as follows,

$$
\begin{aligned}
s_{t+1} &= \rho s_t + (1-\rho)(\nabla Q_i(w))^2 \\
m_{t+1} &= \beta m_t + \frac{\eta}{\sqrt{s_{t+1}+\epsilon}} \nabla Q_i(w) \\
w &= w - m_{t+1}
\end{aligned}
$$

where w represents *var*, which will be updated. s_{t+1} represents *mean_square*, s_t is the last moment of s_{t+1}, m_{t+1} represents *moment*, m_t is the last moment of m_{t+1}. ρ represents *decay*. β is the momentum term, represents *momentum*. ϵ is a smoothing term to avoid division by zero, represents *epsilon*. η represents *learning_rate*. $\nabla Q_i(w)$ represents *grad*.

Warning: Note that in dense implementation of this algorithm, *mean_square* and *moment* will update even if *grad* is 0, but in this sparse implementation, *mean_square* and *moment* will not update in iterations during which *grad* is 0.

Parameters

use_locking (*bool, optional*) – Whether to enable a lock to protect the variable and accumulation tensors from being updated. Default: `False` .

Inputs:

- **var** (Parameter) - Weights to be updated.
- **mean_square** (Tensor) - Mean square gradients, must be the same type as *var*.
- **moment** (Tensor) - Delta of *var*, must be the same type as *var*.
- **learning_rate** (Union[Number, Tensor]) - Learning rate. Must be a float number or a scalar tensor with float16 or float32 data type.
- **grad** (Tensor) - Gradient, must be the same type as *var*.
- **decay** (float) - Decay rate. Only constant value is allowed.
- **momentum** (float) - Momentum. Only constant value is allowed.
- **epsilon** (float) - Ridge term. Only constant value is allowed.

Outputs:

Tensor, parameters to be updated.

Raises

- **TypeError** –If *use_locking* is not a bool.
- **TypeError** –If *var*, *mean_square*, *moment* or *decay* is not a Tensor.
- **TypeError** –If *learning_rate* is neither a Number nor a Tensor.
- **TypeError** –If dtype of *decay*, *momentum* or *epsilon* is not float.
- **TypeError** –If dtype of *learning_rate* is neither float16 nor float32.
- **ValueError** –If *decay*, *momentum* or *epsilon* is not a constant value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.apply_rms_prop = ops.ApplyRMSProp()
...         self.var = Parameter(Tensor(np.ones([2, 2]).astype(np.float32)), name="var")
...
...     def construct(self, mean_square, moment, grad, decay, momentum, epsilon, lr):
...         out = self.apply_rms_prop(self.var, mean_square, moment, lr, grad, decay,␣
↪momentum, epsilon)
...         return out
...
>>> net = Net()
>>> mean_square = Tensor(np.ones([2, 2]).astype(np.float32))
>>> moment = Tensor(np.ones([2, 2]).astype(np.float32))
>>> grad = Tensor(np.ones([2, 2]).astype(np.float32))
>>> output = net(mean_square, moment, grad, 0.0, 1e-10, 0.001, 0.01)
>>> print(net.var.asnumpy())
[[0.990005  0.990005]
 [0.990005  0.990005]]
```

mindspore.ops.LARSUpdate

class mindspore.ops.**LARSUpdate**(*epsilon=1e-05*, *hyperpara=0.001*, *use_clip=False*)

Conducts LARS (layer-wise adaptive rate scaling) update on the sum of squares of gradient.

For more details, please refer to *mindspore.nn.LARS*.

Parameters

- **epsilon** (*float, optional*) –Term added to the denominator to improve numerical stability. Default: `1e-05`.
- **hyperpara** (*float, optional*) –Trust coefficient for calculating the local learning rate. Default: `0.001`.
- **use_clip** (*bool, optional*) –Whether to use clip operation for calculating the local learning rate. Default: `False`.

Inputs:

- **weight** (Tensor) - A tensor, representing the weight. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **gradient** (Tensor) - The gradient of weight, which has the same shape and dtype with weight.
- **norm_weight** (Tensor) - A scalar tensor, representing the sum of squares of weight.
- **norm_gradient** (Tensor) - A scalar tensor, representing the sum of squares of gradient.
- **weight_decay** (Union[Number, Tensor]) - Weight decay. It must be a scalar tensor or number.
- **learning_rate** (Union[Number, Tensor]) - Learning rate. It must be a scalar tensor or number.

Outputs:

Tensor, represents the new gradient.

Raises

- **TypeError** –If neither *epsilon* nor *hyperpara* is a float.
- **TypeError** –If *use_clip* is not a bool.
- **TypeError** –If *weight*, *gradient*, *norm_weight* or *norm_gradient* is not a Tensor.
- **TypeError** –If *weight_decay* or *learning_rate* is neither a Number nor a Tensor.
- **TypeError** –If shape of *gradient* is not the same as *weight*.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.lars = ops.LARSUpdate()
...         self.reduce = ops.ReduceSum()
...         self.square = ops.Square()
...     def construct(self, weight, gradient):
...         w_square_sum = self.reduce(self.square(weight))
...         grad_square_sum = self.reduce(self.square(gradient))
...         grad_t = self.lars(weight, gradient, w_square_sum, grad_square_sum, 0.0, 1.0)
...         return grad_t
...
>>> weight = Tensor(np.array([[0.5, 0.8, 0.2], [0.6, 0.4, 0.2]]).astype(np.float32))
>>> gradient = Tensor(np.array([[0.4, 0.4, 0.5], [0.2, 0.4, 0.3]]).astype(np.float32))
>>> net = Net()
>>> output = net(Tensor(weight), Tensor(gradient))
>>> print(output)
[[0.0005265  0.0005265 0.00065813]
 [0.00026325 0.0005265 0.00039488]]
```

mindspore.ops.SparseApplyAdagradV2

class mindspore.ops.**SparseApplyAdagradV2**(*lr*, *epsilon*, *use_locking=False*, *update_slots=True*)

Updates relevant entries according to the adagrad scheme, one more epsilon attribute than SparseApplyAdagrad.

$$
\begin{aligned}
accum+ &= grad * grad \\
var- &= lr * grad * \frac{1}{\sqrt{accum+\epsilon}}
\end{aligned}
$$

where ϵ represents *epsilon*.

Inputs of *var*, *accum* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

- **lr** (*float*) –Learning rate.
- **epsilon** (*float*) –A small value added for numerical stability.
- **use_locking** (*bool, optional*) –If `True` , the *var* and *accum* tensors will be protected from being updated. Default: `False` .
- **update_slots** (*bool, optional*) –If `True` , the computation logic will be different to *False*. Default: `True` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - Accumulation to be updated. The shape must be the same as *var*.
- **grad** (Tensor) - Gradients has the same shape as *var* and $grad.shape[1:] = var.shape[1:]$ if var.shape > 1.
- **indices** (Tensor) - A vector of indices into the first dimension of *var* and *accum*. The type must be int32 and $indices.shape[0] = grad.shape[0]$. The value of indices must be unique. Otherwise, the result is unpredictable.

Outputs:

Tuple of 2 tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.

Raises

- **TypeError** –If neither *lr* nor *epsilon* is a float.
- **TypeError** –If neither *update_slots* nor *use_locking* is a bool.
- **TypeError** –If dtype of *var*, *accum* or *grad* is neither float16 nor float32.
- **TypeError** –If dtype of *indices* is not int32.
- **RuntimeError** –If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.sparse_apply_adagrad_v2 = ops.SparseApplyAdagradV2(lr=1e-8, epsilon=1e-6)
...         self.var = Parameter(Tensor(np.array([[0.2]]).astype(np.float32)), name="var")
...         self.accum = Parameter(Tensor(np.array([[0.1]]).astype(np.float32)), name="accum
 ↪")
...
...     def construct(self, grad, indices):
...         out = self.sparse_apply_adagrad_v2(self.var, self.accum, grad, indices)
...         return out
...
>>> net = Net()
>>> grad = Tensor(np.array([[0.7]]).astype(np.float32))
>>> indices = Tensor(np.array([0]), mindspore.int32)
>>> output = net(grad, indices)
>>> print(output)
(Tensor(shape=[1, 1], dtype=Float32, value=
[[ 1.99999988e-01]]), Tensor(shape=[1, 1], dtype=Float32, value=
[[ 5.89999974e-01]]))
```

mindspore.ops.SparseApplyProximalAdagrad

class mindspore.ops.**SparseApplyProximalAdagrad**(*use_locking=False*)

Updates relevant entries according to the proximal adagrad algorithm. Compared with *mindspore.ops.ApplyProximalAdagrad*, an additional index tensor is input.

$$
\begin{aligned}
&accum += grad * grad \\
&\text{prox\_v} = var - lr * grad * \frac{1}{\sqrt{accum}} \\
&var = \frac{sign(\text{prox\_v})}{1 + lr * l2} * \max(|\text{prox\_v}| - lr * l1, 0)
\end{aligned}
$$

Inputs of *var*, *accum* and *grad* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool*) – If `True` , the *var* and *accum* tensors will be protected from being updated. Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - Variable tensor to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Parameterv) - Variable tensor to be updated, has the same shape as *var*.
- **lr** (Union[Number, Tensor]) - The learning rate value, must be a float number or a scalar tensor with float16 or float32 data type. It must be positive.
- **l1** (Union[Number, Tensor]) - l1 regularization strength, must be a float number or a scalar tensor with float16 or float32 data type. It must be non-negative.

- **l2** (Union[Number, Tensor]) - l2 regularization strength, must be a float number or a scalar tensor with float16 or float32 data type. It must be non-negative.
- **grad** (Tensor) - A tensor must meet with $grad.shape[1:] = var.shape[1:]$ if var.shape > 1.
- **indices** (Tensor) - A tensor of indices in the first dimension of *var* and *accum*. If there are duplicates in *indices*, the behavior is undefined. Must be one of the following types: int32, int64 and $indices.shape[0] = grad.shape[0]$.

Outputs:

Tuple of 2 tensors, the updated parameters or tensors.

- **var** (Tensor) - The same shape and data type as *var*.
- **accum** (Tensor) - The same shape and data type as *accum*.

Raises

- **TypeError** –If *use_locking* is not a bool.
- **TypeError** –If dtype of *var*, *accum*, *lr*, *l1*, *l2* or *grad* is neither float16 nor float32.
- **TypeError** –If dtype of *indices* is neither int32 nor int64.
- **ValueError** –If *lr* <= 0 or *l1* < 0 or *l2* < 0.
- **RuntimeError** –If the data type of *var*, *accum* and *grad* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.sparse_apply_proximal_adagrad = ops.SparseApplyProximalAdagrad()
...         self.var = Parameter(Tensor(np.array([[4.1, 7.2], [1.1, 3.0]], np.float32)),␣
↪name="var")
...         self.accum = Parameter(Tensor(np.array([[0, 0], [0, 0]], np.float32)), name=
↪"accum")
...         self.lr = 1.0
...         self.l1 = 1.0
...         self.l2 = 0.0
...     def construct(self, grad, indices):
...         out = self.sparse_apply_proximal_adagrad(self.var, self.accum, self.lr, self.l1,
...                                                  self.l2, grad, indices)
...         return out
...
>>> net = Net()
>>> grad = Tensor(np.array([[1, 1], [1, 1]], np.float32))
>>> indices = Tensor(np.array([0, 1], np.int32))
>>> output = net(grad, indices)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Float32, value=
[[ 2.09999990e+00,  5.19999981e+00],
 [ 0.00000000e+00,  1.00000000e+00]]), Tensor(shape=[2, 2], dtype=Float32, value=
```

(continues on next page)

(continued from previous page)

```
[[ 1.00000000e+00,  1.00000000e+00],
 [ 1.00000000e+00,  1.00000000e+00]]))
```

mindspore.ops.SGD

class mindspore.ops.**SGD**(*dampening=0.0*, *weight_decay=0.0*, *nesterov=False*)

Computes the stochastic gradient descent. Momentum is optional.

Nesterov momentum is based on the formula from paper On the importance of initialization and momentum in deep learning.

Note: If parameters are not grouped, the *weight_decay* in optimizer will be applied on the network parameters without 'beta' or 'gamma' in their names. Users can group parameters to change the strategy of decaying weight. When parameters are grouped, each group can set *weight_decay*. If not, the *weight_decay* in optimizer will be applied. For more details, please refer to *mindspore.nn.SGD*.

Parameters

- **dampening** (*float*) – The dampening for momentum. Default: 0.0 .
- **weight_decay** (*float*) – Weight decay (L2 penalty). Default: 0.0 .
- **nesterov** (*bool*) – Enable Nesterov momentum. Default: False .

Inputs:

- **parameters** (Tensor) - Parameters to be updated. With float16 or float32 data type.
- **gradient** (Tensor) - Gradient, with float16 or float32 data type.
- **learning_rate** (Tensor) - Learning rate, a scalar tensor with float16 or float32 data type. e.g. Tensor(0.1, mindspore.float32)
- **accum** (Tensor) - Accum(velocity) to be updated. With float16 or float32 data type.
- **momentum** (Tensor) - Momentum, a scalar tensor with float16 or float32 data type. e.g. Tensor(0.1, mindspore.float32).
- **stat** (Tensor) - States to be updated with the same shape as gradient, with float16 or float32 data type.

Outputs:

Tensor, parameters to be updated.

Raises

- **TypeError** – If *dampening* or *weight_decay* is not a float.
- **TypeError** – If *nesterov* is not a bool.
- **TypeError** – If *parameters*, *gradient*, *learning_rate*, *accum*, *momentum* or *stat* is not a Tensor.
- **TypeError** – If dtype of *parameters*, *gradient*, *learning_rate*, *accum*, *momentum* or *stat* is neither float16 nor float32.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> sgd = ops.SGD()
>>> parameters = Tensor(np.array([2, -0.5, 1.7, 4]), mindspore.float32)
>>> gradient = Tensor(np.array([1, -1, 0.5, 2]), mindspore.float32)
>>> learning_rate = Tensor(0.01, mindspore.float32)
>>> accum = Tensor(np.array([0.1, 0.3, -0.2, -0.1]), mindspore.float32)
>>> momentum = Tensor(0.1, mindspore.float32)
>>> stat = Tensor(np.array([1.5, -0.3, 0.2, -0.7]), mindspore.float32)
>>> output = sgd(parameters, gradient, learning_rate, accum, momentum, stat)
>>> print(output.asnumpy())
[1.99 -0.4903 1.695 3.9801]
```

mindspore.ops.SparseApplyFtrl

class mindspore.ops.**SparseApplyFtrl**(*lr*, *l1*, *l2*, *lr_power*, *use_locking=False*)

Updates relevant entries according to the FTRL-proximal scheme For more details, please refer to *mindspore.nn.FTRL*.

All of inputs except *indices* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

- **lr** (*float*) –The learning rate value, must be positive.
- **l1** (*float*) –l1 regularization strength, must be greater than or equal to zero.
- **l2** (*float*) –l2 regularization strength, must be greater than or equal to zero.
- **lr_power** (*float*) –Learning rate power controls how the learning rate decreases during training, must be less than or equal to zero. Use fixed learning rate if *lr_power* is zero.
- **use_locking** (*bool, optional*) –Use locks for updating operation if `True` . Default: `False` .

Inputs:

- **var** (Union[Parameter, Tensor]) - The variable to be updated. The data type must be float16 or float32. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **accum** (Union[Parameter, Tensor]) - The accumulation to be updated, must be same shape as *var*.
- **linear** (Union[Parameter, Tensor]) - The linear coefficient to be updated, must be the same shape as *var*.
- **grad** (Tensor) - A tensor must meet with $grad.shape[1:] = var.shape[1:]$ if var.shape > 1.
- **indices** (Tensor) - A tensor of indices in the first dimension of *var* and *accum*. If there are duplicates in *indices*, the behavior is undefined. The type must be int32 or int64 and $indices.shape[0] = grad.shape[0]$.

Outputs:

- **var** (Tensor) - Tensor, has the same shape and data type as *var*.
- **accum** (Tensor) - Tensor, has the same shape and data type as *accum*.
- **linear** (Tensor) - Tensor, has the same shape and data type as *linear*.

Raises

- **TypeError** –If *lr*, *l1*, *l2* or *lr_power* is not a float.

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If dtype of *var*, *accum*, *linear* or *grad* is neither float16 nor float32.
- **TypeError** – If dtype of *indices* is neither int32 nor int64.
- **RuntimeError** – If the data type of all of inputs except *indices* conversion of Parameter is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, Parameter, ops
>>> class SparseApplyFtrlNet(nn.Cell):
...     def __init__(self):
...         super(SparseApplyFtrlNet, self).__init__()
...         self.sparse_apply_ftrl = ops.SparseApplyFtrl(lr=0.01, l1=0.0, l2=0.0, lr_power=-
↪0.5)
...         self.var = Parameter(Tensor(np.array([[0.2]]).astype(np.float32)), name="var")
...         self.accum = Parameter(Tensor(np.array([[0.1]]).astype(np.float32)), name="accum
↪")
...         self.linear = Parameter(Tensor(np.array([[0.6]]).astype(np.float32)), name=
↪"linear")
...
...     def construct(self, grad, indices):
...         out = self.sparse_apply_ftrl(self.var, self.accum, self.linear, grad, indices)
...         return out
...
>>> net = SparseApplyFtrlNet()
>>> grad = Tensor(np.array([[0.7]]).astype(np.float32))
>>> indices = Tensor(np.ones([1]), mindspore.int32)
>>> output = net(grad, indices)
>>> print(output)
(Tensor(shape=[1, 1], dtype=Float32, value=
[[2.00000003e-01]]), Tensor(shape=[1, 1], dtype=Float32, value=
[[1.00000001e-01]]), Tensor(shape=[1, 1], dtype=Float32, value=
[[6.00000024e-01]]))
```

mindspore.ops.SparseApplyFtrlV2

class mindspore.ops.**SparseApplyFtrlV2**(*lr*, *l1*, *l2*, *l2_shrinkage*, *lr_power*, *use_locking=False*)

The SparseApplyFtrlV2 interface is deprecated, please use the *mindspore.ops.SparseApplyFtrl* instead.

Supported Platforms:

Deprecated

18.2.5 Distance Function

API Name	Description	Supported Platforms
mindspore.ops.Cdist	Refer to *mindspore.ops.cdist()* for more details.	Ascend GPU CPU
mindspore.ops.EditDistance	Computes the Levenshtein Edit Distance.	Ascend CPU
mindspore.ops.LpNorm	Return the p-norm of a matrix or vector.	Ascend GPU CPU
mindspore.ops.Pdist	Computes the p-norm distance between each pair of row vectors in the input.	GPU CPU

mindspore.ops.Cdist

class mindspore.ops.**Cdist**(*p=2.0*)

```
prim = ops.Cdist(p)
out = prim(x1, x2)
```

is equivalent to

```
ops.cdist(x1, x2, p)
```

Refer to *mindspore.ops.cdist()* for more details.

mindspore.ops.EditDistance

class mindspore.ops.**EditDistance**(*normalize=True*)

Computes the Levenshtein Edit Distance. It is used to measure the similarity of two sequences. The inputs are variable-length sequences provided by SparseTensors (hypothesis_indices, hypothesis_values, hypothesis_shape) and (truth_indices, truth_values, truth_shape).

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0 \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Where the a indicates the hypothesis and the b indicates the truth. For ease of understanding, i and j here in may be considered as lengths of a and b.

> **Warning:** Unorded *truth_indices* or *hypothesis_indices* might lead to expected result, so it is suggested to make sure *truth_indices* and *hypothesis_indices* are both in ascending order before calling this API.

Parameters

normalize (*bool*) – If `True` , edit distances are normalized by length of truth. Default: `True` .

Inputs:

- **hypothesis_indices** (Tensor) - The indices of the hypothesis list SparseTensor. With int64 data type. The shape of tensor is (N, R).
- **hypothesis_values** (Tensor) - The values of the hypothesis list SparseTensor. Must be 1-D vector with length of N.

- **hypothesis_shape** (Tensor) - The shape of the hypothesis list SparseTensor. Must be R-length vector with int64 data type. Only constant value is allowed.
- **truth_indices** (Tensor) - The indices of the truth list SparseTensor. With int64 data type. The shape of tensor is (M, R).
- **truth_values** (Tensor) - The values of the truth list SparseTensor. Must be 1-D vector with length of M.
- **truth_shape** (Tensor) - The shape of the truth list SparseTensor. Must be R-length vector with int64 data type. Only constant value is allowed.

Outputs:

Tensor, a dense tensor with rank *R-1* and float32 data type.

Raises

TypeError – If *normalize* is not a bool.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> class EditDistance(nn.Cell):
...     def __init__(self, hypothesis_shape, truth_shape, normalize=True):
...         super(EditDistance, self).__init__()
...         self.edit_distance = ops.EditDistance(normalize)
...         self.hypothesis_shape = hypothesis_shape
...         self.truth_shape = truth_shape
...
...     def construct(self, hypothesis_indices, hypothesis_values, truth_indices, truth_
↪values):
...         return self.edit_distance(hypothesis_indices, hypothesis_values, self.hypothesis_
↪shape,
...                                   truth_indices, truth_values, self.truth_shape)
...
>>> hypothesis_indices = Tensor(np.array([[0, 0, 0], [1, 0, 1], [1, 1, 1]]).astype(np.int64))
>>> hypothesis_values = Tensor(np.array([1, 2, 3]).astype(np.float32))
>>> hypothesis_shape = Tensor(np.array([1, 1, 2]).astype(np.int64))
>>> truth_indices = Tensor(np.array([[0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1]]).astype(np.
↪int64))
>>> truth_values = Tensor(np.array([1, 3, 2, 1]).astype(np.float32))
>>> truth_shape = Tensor(np.array([2, 2, 2]).astype(np.int64))
>>> edit_distance = EditDistance(hypothesis_shape, truth_shape)
>>> output = edit_distance(hypothesis_indices, hypothesis_values, truth_indices, truth_
↪values)
>>> print(output)
[[1. 1.]
 [1. 1.]]
```

mindspore.ops.LpNorm

class mindspore.ops.**LpNorm**(*axis*, *p=2*, *keep_dims=False*, *epsilon=1e-12*)

Return the p-norm of a matrix or vector.

$$output = \|input\|_p = \left(\sum_{i=1}^{n} |input|^p\right)^{1/p}$$

Parameters

- **axis** (*int*, *list*, *tuple*) – Specifies which dimension or dimensions of input to calculate the norm across.
- **p** (*int*, *optional*) – The order of norm. Default: `2` .
- **keep_dims** (*bool*, *optional*) – Whether the output tensors have dim retained or not. Default: `False` .
- **epsilon** (*float*, *optional*) – The lower bound value, when the calculated norm is less than this value, replace this result with *epsilon*. Default: `1e-12` .

Inputs:

- **input** (Tensor) - Input tensor of type float16, float32.

Outputs:

Tensor, has the same dtype as *input*, its shape depends on *axis*. For example, if the shape of input is $(2, 3, 4)$, *axis* is $[0, 1]$, output shape will be $(4,)$.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If dtype of *input* is not one of: float16, float32.
- **TypeError** – If *p* is not an int.
- **TypeError** – If *axis* is not an int, a tuple or a list.
- **TypeError** – If *axis* is a tuple or a list, but the element of *axis* is not an int.
- **TypeError** – If *keep_dims* is not a bool.
- **ValueError** – If the element of *axis* is out of the range $[-r, r)$, where r is the rank of *input*.
- **ValueError** – If the length of shape of *axis* is bigger than the length of shape of *input*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[[1.0, 2.0], [3.0, 4.0]], [[5.0, 6.0], [7.0, 8.0]]]).
↪astype(np.float32))
>>> op = ops.LpNorm(axis=[0, 1], p=2, keep_dims=False)
>>> output = op(input_x)
>>> print(output)
[ 9.165152 10.954452]
```

mindspore.ops.Pdist

class mindspore.ops.**Pdist**(*p=2.0*)

Computes the p-norm distance between each pair of row vectors in the input.

Refer to *mindspore.ops.pdist()* for more details.

Note: The pdist operator involves exponentiation, the inf/nan calculation result may be generated when the float16 input is used. The float32 input is recommended.

Parameters

p (*float, optional*) – The order of norm distance, $p \in [0,)$. Default: `2.0`.

Inputs:

- **x** (Tensor) - Input tensor. Supported dtypes: float16, float32 or float64.

Outputs:

Tensor, has the same dtype as *x*.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> x = Tensor(np.array([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]]).astype(np.float32))
>>> op = ops.Pdist(p=2.0)
>>> y = op(x)
>>> print(y)
[1.4142135 2.828427  1.4142135]
```

18.2.6 Sampling Operator

API Name	Description	Supported Platforms
mindspore.ops.ComputeAccidentalHits	Compute accidental hits of sampled classes which match target classes.	`Ascend`
mindspore.ops.LogUniformCandidateSampler	Generates random labels with a log-uniform distribution for sampled_candidates.	`Ascend` `CPU`
mindspore.ops.UniformCandidateSampler	Uniform candidate sampler.	`Ascend` `GPU` `CPU`

mindspore.ops.ComputeAccidentalHits

class mindspore.ops.**ComputeAccidentalHits**(*num_true=1*)

Compute accidental hits of sampled classes which match target classes.

When a target class matches the sample class, we call it "accidental hit". The result of calculating accidental hits contain three parts (index, id, weight), where index represents the row number in true_classes, and id represents the position in sampled_candidates, the weight is FLOAT_MAX. FLOAT_MAX indicates the max value in the type of Float

Parameters

num_true (*int, optional*) – The number of target classes per training example. Default: `1` .

Inputs:

- **true_classes** (Tensor) - The target classes. With data type of int64 and shape $(batch\_size, num\_true)$.
- **sampled_candidates** (Tensor) - The Candidate sampling results of operators, types of training samples, with data type of int64 and shape $(num\_sampled,)$.

Outputs:

Tuple of 3 Tensors.

- **indices** (Tensor) - A Tensor with shape $(num\_accidental\_hits,)$, with data type of int32.
- **ids** (Tensor) - A Tensor with shape $(num\_accidental\_hits,)$, with data type of int64.
- **weights** (Tensor) - A Tensor with shape $(num\_accidental\_hits,)$, with the type float32.

Raises

- **TypeError** – If dtype of *num_true* is not int.
- **TypeError** – If *true_classes* or *sampled_candidates* is not a Tensor.
- **TypeError** – If dtype of *true_classes* or *sampled_candidates* is neither int32 nor int64.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> true_classes = np.array([[1, 2], [0, 4], [3, 3]])
>>> sampled_candidates = np.array([0, 1, 2, 3, 4])
>>> sampler = ops.ComputeAccidentalHits(2)
>>> indices, ids, weights = sampler(Tensor(true_classes), Tensor(sampled_candidates))
>>> print(indices, ids, weights)
[0 0 1 1 2 2]
[1 2 0 4 3 3]
[-3.4028235e+38 -3.4028235e+38 -3.4028235e+38 -3.4028235e+38 -3.4028235e+38 -3.4028235e+38]
```

mindspore.ops.LogUniformCandidateSampler

class mindspore.ops.**LogUniformCandidateSampler**(*num_true=1*, *num_sampled=5*, *unique=True*, *range_max=5*, *seed=0*)

Generates random labels with a log-uniform distribution for sampled_candidates.

Randomly samples a tensor of sampled classes from the range of integers [0, range_max).

Refer to *mindspore.ops.log _uniform_candidate_sampler()* for more details.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

- **num_true** (*int, optional*) –The number of target classes per training example. Default: `1` .
- **num_sampled** (*int, optional*) –The number of classes to randomly sample. Default: `5` .
- **unique** (*bool, optional*) –Determines whether sample with rejection. If *unique* is `True` , all sampled classes in a batch are unique. Default: `True` .
- **range_max** (*int, optional*) –The number of possible classes. When *unique* is `True` , *range_max* must be greater than or equal to *num_sampled*. Default: `5` .
- **seed** (*int, optional*) –Random seed, must be non-negative. Default: `0` .

Inputs:

- **true_classes** (Tensor) - The target classes. With data type of int64 and shape $(batch\_size, num\_true)$.

Outputs:

Tuple of 3 Tensors.

- **sampled_candidates** (Tensor) - A Tensor with shape $(num\_sampled,)$ and the same type as *true_classes*.
- **true_expected_count** (Tensor) - A Tensor with the same shape as *true_classes and* type float32.
- **sampled_expected_count** (Tensor) - A Tensor with the same shape as *sampled_candidates* and type float32.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> sampler = ops.LogUniformCandidateSampler(2, 5, True, 5)
>>> output1, output2, output3 = sampler(Tensor(np.array([[1, 7], [0, 4], [3, 3]])))
>>> print(output1, output2, output3)
[3 2 0 4 1]
[[0.92312991 0.49336370]
 [0.99248987 0.65806371]
 [0.73553443 0.73553443]]
[0.73553443 0.82625800 0.99248987 0.65806371 0.92312991]
```

mindspore.ops.UniformCandidateSampler

class mindspore.ops.**UniformCandidateSampler**(*num_true*, *num_sampled*, *unique*, *range_max*, *seed=0*, *remove_accidental_hits=False*)

Uniform candidate sampler.

This function samples a set of classes(sampled_candidates) from [0, range_max-1] based on uniform distribution.

Refer to *mindspore.ops.uniform _candidate_sampler()* for more details.

Warning:

- The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.
- The Ascend backend does not support dynamic shape scenarios currently.

Parameters

- **num_true** (*int*) −The number of target classes in each training example.
- **num_sampled** (*int*) −The number of classes to randomly sample. The sampled_candidates will have a shape of num_sampled. If unique=True, num_sampled must be less than or equal to range_max.
- **unique** (*bool*) −Whether all sampled classes in a batch are unique.
- **range_max** (*int*) −The number of possible classes, must be non-negative.
- **seed** (*int, optional*) −Used for random number generation, must be non-negative. If seed has a value of 0, the seed will be replaced with a randomly generated value. Default: `0` .
- **remove_accidental_hits** (*bool, optional*) −Whether accidental hit is removed. Accidental hit is when one of the true classes matches one of the sample classes. Set `True` to remove which accidentally sampling the true class as sample class. Default: `False` .

Inputs:

- **true_classes** (Tensor) - A Tensor. The target classes with a Tensor shape of $(batch\_size, num\_true)$. The value range of the elements must be $[0, range\_max)$.

Outputs:

- **sampled_candidates** (Tensor) - The sampled_candidates is independent of the true classes. Shape: $(num\_sampled,)$.
- **true_expected_count** (Tensor) - The expected counts under the sampling distribution of each of true_classes. Shape: $(batch\_size, num\_true)$.
- **sampled_expected_count** (Tensor) - The expected counts under the sampling distribution of each of sampled_candidates. Shape: $(num\_sampled,)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> sampler = ops.UniformCandidateSampler(1, 3, False, 4, 1)
>>> output1, output2, output3 = sampler(Tensor(np.array([[1], [3], [4], [6], [3]], dtype=np.
↪int64)))
>>> print(output1.shape)
(3,)
>>> print(output2.shape)
(5, 1)
>>> print(output3.shape)
(3,)
```

18.2.7 Image Processing

API Name	Description	Supported Platforms
mindspore.ops.BoundingBoxDecode	Decodes bounding boxes locations.	Ascend GPU CPU
mindspore.ops.BoundingBoxEncode	Encodes bounding boxes locations.	Ascend GPU CPU
mindspore.ops.CheckValid	Checks bounding box.	Ascend GPU CPU
mindspore.ops.CropAndResize	Extracts crops from the input image tensor and resizes them.	Ascend GPU CPU
mindspore.ops.ExtractVolumePatches	*ops.ExtractVolumePatches* is deprecated from version 2.3 and will be removed in a future version.	Deprecated
mindspore.ops.IOU	Calculates intersection over union for boxes.	Ascend GPU CPU
mindspore.ops.L2Normalize	L2 Normalization Operator.	Ascend GPU CPU
mindspore.ops.NMSWithMask	Non-maximum Suppression.	Ascend GPU CPU
mindspore.ops.ResizeBilinearV2	Resizes an image to a certain size using the bilinear interpolation.	Ascend GPU CPU
mindspore.ops.ROIAlign	Computes the Region of Interest (RoI) Align operator.	Ascend GPU CPU
mindspore.ops.UpsampleNearest3D	Performs nearest neighbor upsampling operation.	Ascend GPU CPU
mindspore.ops.UpsampleTrilinear3D	Performs upsampling with trilinear interpolation across 3dims for 5dim input Tensor.	Ascend GPU CPU

mindspore.ops.BoundingBoxDecode

class mindspore.ops.**BoundingBoxDecode**(*max_shape*, *means=(0.0, 0.0, 0.0, 0.0)*, *stds=(1.0, 1.0, 1.0, 1.0)*, *wh_ratio_clip=0.016*)

Decodes bounding boxes locations.

The function of the operator is to calculate the offset, and this operator converts the offset into a Bbox, which is used to mark the target in the subsequent images, etc.

Parameters

- **max_shape** (*tuple*) –The max size limit for decoding box calculation.
- **means** (*tuple, optional*) –The means of deltas calculation. Default: `(0.0, 0.0, 0.0, 0.0)`.
- **stds** (*tuple, optional*) –The standard deviations of deltas calculation. Default: `(1.0, 1.0, 1.0, 1.0)`.

- **wh_ratio_clip** (*float, optional*) –The limit of width and height ratio for decoding box calculation. Default: `0.016`.

Inputs:

- **anchor_box** (Tensor) - Anchor boxes. The shape of *anchor_box* must be $(n, 4)$.
- **deltas** (Tensor) - Delta of boxes. Which has the same shape with *anchor_box*.

Outputs:

Tensor, decoded boxes. It has the same data type and shape as *anchor_box*.

Raises

- **TypeError** –If *means*, *stds* or *max_shape* is not a tuple.
- **TypeError** –If *wh_ratio_clip* is not a float.
- **TypeError** –If *anchor_box* or *deltas* is not a Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> anchor_box = Tensor([[4, 1, 2, 1], [2, 2, 2, 3]], mindspore.float32)
>>> deltas = Tensor([[3, 1, 2, 2], [1, 2, 1, 4]], mindspore.float32)
>>> boundingbox_decode = ops.BoundingBoxDecode(means=(0.0, 0.0, 0.0, 0.0), stds=(1.0, 1.0, 1.
↪0, 1.0),
...                                            max_shape=(768, 1280), wh_ratio_clip=0.016)
>>> output = boundingbox_decode(anchor_box, deltas)
>>> print(output)
[[ 4.194528  0.         0.         5.194528]
 [ 2.1408591   0.         3.8591409  60.598152  ]]
```

mindspore.ops.BoundingBoxEncode

class mindspore.ops.**BoundingBoxEncode**(*means=(0.0, 0.0, 0.0, 0.0)*, *stds=(1.0, 1.0, 1.0, 1.0)*)

Encodes bounding boxes locations.

This operator will calculate the offset between the predicted bounding boxes and the real bounding boxes, and this offset will be used as a variable for the loss.

Parameters

- **means** (*tuple*) –Means for encoding bounding boxes calculation. Default: `(0.0, 0.0, 0.0, 0.0)`.
- **stds** (*tuple*) –The standard deviations of deltas calculation. Default: `(1.0, 1.0, 1.0, 1.0)`.

Inputs:

- **anchor_box** (Tensor) - Anchor boxes. The shape of anchor_box must be $(n, 4)$.
- **groundtruth_box** (Tensor) - Ground truth boxes. Which has the same shape with anchor_box.

Outputs:

Tensor, encoded bounding boxes. It has the same data type and shape as input *anchor_box*.

Raises

- **TypeError** – If *means* or *stds* is not a tuple.
- **TypeError** – If *anchor_box* or *groundtruth_box* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> anchor_box = Tensor([[2, 2, 2, 3], [2, 2, 2, 3]], mindspore.float32)
>>> groundtruth_box = Tensor([[1, 2, 1, 4], [1, 2, 1, 4]], mindspore.float32)
>>> boundingbox_encode = ops.BoundingBoxEncode(means=(0.0, 0.0, 0.0, 0.0), stds=(1.0, 1.0, 1.
↪0, 1.0))
>>> output = boundingbox_encode(anchor_box, groundtruth_box)
>>> print(output)
[[ -1.   0.25  0.   0.40551758]
 [ -1.   0.25  0.   0.40551758]]
```

mindspore.ops.CheckValid

class mindspore.ops.**CheckValid**

Checks bounding box.

Checks whether the bounding boxes specified by *bboxes* is valid. Returns True if the box is within borders specified by *img_metas*, False if not.

Inputs:

- **bboxes** (Tensor) - Bounding boxes tensor with shape $(N, 4)$. N indicates the number of bounding boxes, the value "4" indicates "x0", "y0", "x1", and "y1". Data type must be float16 or float32.
- **img_metas** (Tensor) - Raw image size information with the format of $(height, width, ratio)$, specifying the valid boundary $(height * ratio, width * ratio)$. Data type must be float16 or float32.

Outputs:

Tensor, with shape of $(N,)$ and dtype of bool, specifying whether the bounding boxes is in the image. "True" indicates valid, while "False" indicates invalid.

Raises

- **TypeError** – If *bboxes* or *img_metas* is not a Tensor.
- **TypeError** – If dtype of *bboxes* or *img_metas* is neither float16 nor float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.check_valid = ops.CheckValid()
...     def construct(self, x, y):
...         valid_result = self.check_valid(x, y)
...         return valid_result
...
>>> bboxes = Tensor(np.linspace(0, 6, 12).reshape(3, 4), mindspore.float32)
>>> img_metas = Tensor(np.array([2, 1, 3]), mindspore.float32)
>>> net = Net()
>>> output = net(bboxes, img_metas)
>>> print(output)
[ True False False]
```

mindspore.ops.CropAndResize

class mindspore.ops.**CropAndResize**(*method='bilinear'*, *extrapolation_value=0.0*)

Extracts crops from the input image tensor and resizes them.

Note: In case that the output shape depends on crop_size, the crop_size must be constant. For now, the backward of the operator only support bilinear method, for other methods, will return 0.

Parameters

- **method** (*str, optional*) – An optional string that specifies the sampling method for resizing. It can be "bilinear", "nearest" or "bilinear_v2". Default: "bilinear".
  - "nearest": Nearest neighbor interpolation. Each output pixel is assigned the value of the nearest input pixel. This method is simple and fast but can result in blocky or pixelated outputs.
  - "bilinear": Bilinear interpolation. Each output pixel is a weighted average of the four nearest input pixels, computed using bilinear interpolation. This method produces smoother results compared to nearest neighbor interpolation.
  - "bilinear_v2": The optimized variant of "bilinear", it may achieve better result(higher precision and speed) in some cases.
- **extrapolation_value** (*float, optional*) – An optional float value used extrapolation, if applicable. Default: 0.0.

Inputs:

- **x** (Tensor) - The input image must be a 4-D tensor of shape $(batch, image\_height, image\_width, depth)$. Types allowed: int8, int16, int32, int64, float16, float32, float64, uint8, uint16.
- **boxes** (Tensor) - A 2-D tensor of shape $(num\_boxes, 4)$. The i-th row of the tensor specifies the coordinates of a box in the box_ind[i] image and is specified in normalized coordinates [y1, x1, y2, x2]. A normalized coordinate value of y is mapped to the image coordinate at y * (image_height - 1), so as the [0, 1] interval of normalized image height is mapped to [0, image_height - 1] in image height coordinates. We do allow y1 > y2, in which case the sampled crop is

an up-down flipped version of the original image. The width dimension is treated similarly. Normalized coordinates outside the [0, 1] range are allowed, in which case we use *extrapolation_value* to extrapolate the input image values. Types allowed: float32.

- **box_index** (Tensor) - A 1-D tensor of shape (num_boxes) with int32 values in [0, batch). The value of *box_index[i]* specifies the image that the i-th box refers to. Types allowed: int32.
- **crop_size** (Tuple[int]) - A tuple of two int32 elements: (crop_height, crop_width). Only constant value is allowed. All cropped image patches are resized to this size. The aspect ratio of the image content is not preserved. Both crop_height and crop_width need to be positive.

Outputs:

A 4-D tensor of shape ($num\_boxes, crop\_height, crop\_width, depth$) with type: float32.

Raises

- **TypeError** – If *x* or *boxes* or *box_index* is not a Tensor.
- **TypeError** – If *crop_size* is not a Tuple with two int32 elements.
- **TypeError** – If dtype of *boxes* is not float or that of *box_index* is not int.
- **TypeError** – If *method* is not a str.
- **TypeError** – If *extrapolation_value* is not a float.
- **ValueError** – If the shape rank of *x* is not 4.
- **ValueError** – If the shape rank of *boxes* is not 2.
- **ValueError** – If the second dim of *boxes* is not 4.
- **ValueError** – If the shape rank of *box_index* is not 1.
- **ValueError** – If the first dim of *box_index* is not equal to that of *boxes*.
- **ValueError** – If existing element in *box_index* is out of range *[0, batch)*.
- **ValueError** – If the data of *crop_size* is not positive.
- **ValueError** – If *method* is not one of 'bilinear', 'nearest', 'bilinear_v2'.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import nn, ops, Tensor
>>> class CropAndResizeNet(nn.Cell):
...     def __init__(self, crop_size):
...         super(CropAndResizeNet, self).__init__()
...         self.crop_and_resize = ops.CropAndResize()
...         self.crop_size = crop_size
...
...     def construct(self, x, boxes, box_index):
...         return self.crop_and_resize(x, boxes, box_index, self.crop_size)
...
>>> BATCH_SIZE = 1
>>> NUM_BOXES = 5
```

(continues on next page)

(continued from previous page)

```
>>> IMAGE_HEIGHT = 256
>>> IMAGE_WIDTH = 256
>>> CHANNELS = 3
>>> image = np.random.normal(size=[BATCH_SIZE, IMAGE_HEIGHT, IMAGE_WIDTH, CHANNELS]).
↪astype(np.float32)
>>> boxes = np.random.uniform(size=[NUM_BOXES, 4]).astype(np.float32)
>>> box_index = np.random.uniform(size=[NUM_BOXES], low=0, high=BATCH_SIZE).astype(np.int32)
>>> crop_size = (24, 24)
>>> crop_and_resize = CropAndResizeNet(crop_size=crop_size)
>>> output = crop_and_resize(Tensor(image), Tensor(boxes), Tensor(box_index))
>>> print(output.shape)
(5, 24, 24, 3)
```

mindspore.ops.ExtractVolumePatches

class mindspore.ops.**ExtractVolumePatches**(*kernel_size*, *strides*, *padding*)

ops.ExtractVolumePatches is deprecated from version 2.3 and will be removed in a future version.

Supported Platforms:

Deprecated

mindspore.ops.IOU

class mindspore.ops.**IOU**(*mode='iou'*)

Calculates intersection over union for boxes.

Computes the intersection over union (IOU) or the intersection over foreground (IOF) based on the truth and predicted regions.

Refer to *mindspore.ops.iou()* for more details.

Parameters

mode (*str*) – The mode is used to specify the calculation method, now supporting `'iou'` (intersection over union) or `'iof'` (intersection over foreground) mode. Default: `'iou'` .

Inputs:

- **anchor_boxes** (Tensor) - Anchor boxes, tensor of shape $(N, 4)$. "N" indicates the number of anchor boxes, and the value "4" refers to "x0", "y0", "x1", and "y1". Data type must be float16 or float32.
- **gt_boxes** (Tensor) - The truth boxes, tensor of shape $(M, 4)$. "M" indicates the number of truth boxes, and the value "4" refers to "x0", "y0", "x1", and "y1". Data type must be float16 or float32.

Outputs:

Tensor, the 'iou' values, tensor of shape (M, N), with the same data type as *anchor_boxes*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> iou = ops.IOU(mode='iou')
>>> anchor_boxes = Tensor(np.random.randint(1.0, 5.0, [3, 4]), mindspore.float16)
>>> gt_boxes = Tensor(np.random.randint(1.0, 5.0, [3, 4]), mindspore.float16)
>>> output = iou(anchor_boxes, gt_boxes)
>>> print(output.shape)
(3, 3)
```

mindspore.ops.L2Normalize

class mindspore.ops.**L2Normalize**(*axis=0*, *epsilon=1e-4*)

L2 Normalization Operator.

This operator will normalize the input using the given axis. The function is shown as follows:

$$\text{output} = \frac{x}{\sqrt{\max(\sum_i |x_i|^2, \epsilon)}}$$

where ϵ is epsilon and $\sum_i |x_i|^2$ calculate the sum of squares of the input x along the dimension *axis*.

Note: On Ascend, input data type of float64 is currently not supported.

Parameters

- **axis** (*Union[list(int), tuple(int), int], optional*) – Specify the axis for calculating the L2 norm. Default: `0` .
- **epsilon** (*float, optional*) – A small value added for numerical stability. Default: `1e-4` .

Inputs:

- **x** (Tensor) - Input to compute the normalization. Tensor of shape $(N, *)$, where $*$ means any number of additional dimensions. Data type must be float16, float32 or float64.

Outputs:

Tensor, with the same type and shape as the *x*.

Raises

- **TypeError** – If *axis* is not one of the following: list, tuple or int.
- **TypeError** – If *epsilon* is not a float.
- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If dtype of *x* is not in [float16, float32, float64].
- **ValueError** – If dimension of *x* is not greater than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> l2_normalize = ops.L2Normalize()
>>> x = Tensor(np.random.randint(-256, 256, (2, 3, 4)), mindspore.float32)
>>> output = l2_normalize(x)
>>> print(output.shape)
(2, 3, 4)
```

mindspore.ops.NMSWithMask

class mindspore.ops.**NMSWithMask**(*iou_threshold=0.5*)

Non-maximum Suppression. When object detection problem is performed in the computer vision field, object detection algorithm generates a plurality of bounding boxes. Use the box with the highest score, calculate the overlap between other boxes and the current box, and delete the box based on a certain threshold(IOU). On Ascend platform, the input box score is ignored, which only selects boexs based on the IOU between boxes, which means if you want to remove boxes that has lower scores, you need to sort the input boxes by score in descending order in advance. The IOU is as follows:

$$\text{IOU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

> **Warning:** Only supports up to 2864 input boxes at one time.

Parameters

iou_threshold (*float*) –Specifies the threshold of overlap boxes with respect to IOU. Default: `0.5` .

Inputs:

- **bboxes** (Tensor) - The shape of tensor is $(N, 5)$. Input bounding boxes. N is the number of input bounding boxes. Every bounding box contains 5 values, the first 4 values are the coordinates(x0, y0, x1, y1) of bounding box which represents the point of top-left and bottom-right, and the last value is the score of this bounding box. The data type must be float16 or float32.

Outputs:

tuple[Tensor], tuple of three tensors, they are output_boxes, output_idx and selected_mask.

- **output_boxes** (Tensor) - The shape of tensor is $(N, 5)$. On GPU and CPU platform, it is a sorted list of bounding boxes by sorting the input *bboxes* in descending order of score. On Ascend platform, it is same as input *bboxes*.
- **output_idx** (Tensor) - The shape of tensor is $(N,)$. The indexes list of *output_boxes*.
- **selected_mask** (Tensor) - The shape of tensor is $(N,)$. A mask list of valid output bounding boxes. Apply this mask on *output_boxes* to get the list of bounding boxes after non-max suppression calculation, or apply this mask on *output_idx* to get the indexes list of bounding boxes after non-max suppression calculation.

Raises

- **ValueError** –If the *iou_threshold* is not a float number.
- **ValueError** –if the first dimension of input Tensor is less than or equal to 0.
- **TypeError** –if the dtype of the *bboxes* is not float16 or float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bbox = np.array([[100.0, 100.0, 50.0, 68.0, 0.63], [150.0, 75.0, 165.0, 115.0, 0.55],
...                  [12.0, 190.0, 288.0, 200.0, 0.9], [28.0, 130.0, 106.0, 172.0, 0.3]])
>>> bbox[:, 2] += bbox[:, 0]
>>> bbox[:, 3] += bbox[:, 1]
>>> inputs = Tensor(bbox, mindspore.float32)
>>> nms = ops.NMSWithMask(0.1)
>>> output_boxes, indices, mask = nms(inputs)
>>> indices_np = indices.asnumpy()
>>> print(indices_np[mask.asnumpy()])
[0 1 2]
```

mindspore.ops.ResizeBilinearV2

class mindspore.ops.**ResizeBilinearV2**(*align_corners=False*, *half_pixel_centers=False*)

Resizes an image to a certain size using the bilinear interpolation.

The resizing only affects the lower two dimensions which represent the height and width of the image.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **align_corners** (*bool, optional*) –If `True` , rescale input by $(new\_height-1)/(height-1)$, which exactly aligns the 4 corners of images and resized images. If `False` , rescale by $new\_height/height$. Default: `False` .
- **half_pixel_centers** (*bool, optional*) –Whether half pixel center. If set to `True` , *align_corners* should be `False` . Default: `False` .

Inputs:

- **x** (Tensor) - Image to be resized. Input images must be a 4-D tensor with shape $(batch, channels, height, width)$, with data type of float32 or float16.
- **size** (Union[tuple[int], list[int], Tensor]) - The new size of the images. A tuple or list or Tensor of 2 int elements (new_height, new_width).

Outputs:

Tensor, resized image. 4-D with shape $(batch, channels, new\_height, new\_width)$, with the same data type as input *x*.

Raises

- **TypeError** –If *align_corners* is not a bool.
- **TypeError** –If *half_pixel_centers* is not a bool.
- **TypeError** –If *align_corners* and *half_pixel_centers* are `True` .
- **ValueError** –If *half_pixel_centers* is `True` and device_target is CPU.
- **ValueError** –If dim of *x* is not 4.
- **ValueError** –If *size* is Tensor and its dim is not 1.
- **ValueError** –If *size* contains other than 2 elements.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([[[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]]], mindspore.float32)
>>> output = ops.ResizeBilinearV2()(x, (5, 5))
>>> print(output)
[[[[1. 2. 3. 4. 5.]
   [1. 2. 3. 4. 5.]
   [1. 2. 3. 4. 5.]
   [1. 2. 3. 4. 5.]
   [1. 2. 3. 4. 5.]]]]
```

mindspore.ops.ROIAlign

class mindspore.ops.**ROIAlign**(*pooled_height*, *pooled_width*, *spatial_scale*, *sample_num=2*, *roi_end_mode=1*)

Computes the Region of Interest (RoI) Align operator.

The operator computes the value of each sampling point by bilinear interpolation from the nearby grid points on the feature map. No quantization is performed on any coordinates involved in the RoI, its bins, or the sampling points. The details of (RoI) Align operator are described in Mask R-CNN.

Parameters

- **pooled_height** (*int*) –The output features height.
- **pooled_width** (*int*) –The output features width.
- **spatial_scale** (*float*) –A scaling factor that maps the raw image coordinates to the input feature map coordinates. Suppose the height of a RoI is *ori_h* in the raw image and *fea_h* in the input feature map, the *spatial_scale* must be *fea_h / ori_h*.
- **sample_num** (*int*) –Number of sampling points. Default: `2` .
- **roi_end_mode** (*int*) –Number must be 0 or 1. If roi_end_mode=0, use the legacy implementation. If roi_end_mode=1, end pixel of the roi_box will be shifted by +1*spatial_scale. Default: `1` .

Inputs:

- **features** (Tensor) - The input features, whose shape must be (N, C, H, W), with data type of float16 or float32.

- **rois** (Tensor) - The shape is $(rois\_n, 5)$, with data type of float16 or float32. *rois_n* represents the number of RoI. The size of the second dimension must be *5* and the *5* colunms are $(image\_index, top\_left\_x, top\_left\_y, bottom\_right\_x, bottom\_right\_y)$. *image_index* represents the index of image. *top_left_x* and *top_left_y* represent the *x, y* coordinates of the top left corner of corresponding RoI, respectively. *bottom_right_x* and *bottom_right_y* represent the *x, y* coordinates of the bottom right corner of corresponding RoI, respectively.

Outputs:

Tensor, the shape is $(rois\_n, C, pooled\_height, pooled\_width)$.

Raises

- **TypeError** –If *pooled_height*, *pooled_width*, *sample_num* or *roi_end_mode* is not an int.
- **TypeError** –If *spatial_scale* is not a float.
- **TypeError** –If *features* or *rois* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> features = Tensor(np.array([[[[1., 2.], [3., 4.]]]]), mindspore.float32)
>>> rois = Tensor(np.array([[0, 0.2, 0.3, 0.2, 0.3]]), mindspore.float32)
>>> roi_align = ops.ROIAlign(2, 2, 0.5, 2)
>>> output = roi_align(features, rois)
>>> print(output)
[[[[1.775 2.025]
   [2.275 2.525]]]]
```

mindspore.ops.UpsampleNearest3D

class mindspore.ops.**UpsampleNearest3D**

Performs nearest neighbor upsampling operation.

This operator scale up the volumetric input with specified *output_size* or *scales* factors, using nearest neighbor algorithm.

One of *output_size* or *scales* must be given, and can not specified both at the same time.

Inputs:

- **x** (Tensor) - 5D tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$. Supporting types: [float16, float32, float64].
- **output_size** (Union[tuple[int], list[int]]): A tuple or list of int specifying the output volumetric size. Default: `None`.
- **scales** (Union[tuple[float], list[float]]): A tuple or list of float specifying the upsampling factors. Default: `None`.

Outputs:

- **y** (Tensor) - Upsampled output with the same type as *x* , whose shape is $(N, C, D_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** –When *output_size* is not `None` and *output_size* is not list[int] or tuple[int].

- **TypeError** – When *scales* is not `None` and *scales* is not list[float] or tuple[float].
- **TypeError** – If dtype of *x* is not int [float16, float32, float64].
- **ValueError** – If any value of *output_size* is negative or zero when *output_size* is not `None`.
- **ValueError** – If any value of *scales* is negative or zero when *scales* is not `None`.
- **ValueError** – If shape of *x* is not 5D.
- **ValueError** – If none of *scales* and *output_size* is specified or both specified.
- **ValueError** – If size of *scales* is not equal 3 when *scales* is specified.
- **ValueError** – If size of *output_size* is not equal 3 when *output_size* is specified.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])
...        .reshape([1, 1, 2, 2, 4]), mstype.float32)
>>> output_size = [3, 4, 5]
>>> net = ops.UpsampleNearest3D()
>>> output = net(x, output_size, None)
>>> print(output)
[[[[[ 1.  1.  2.  3.  4.]
    [ 1.  1.  2.  3.  4.]
    [ 5.  5.  6.  7.  8.]
    [ 5.  5.  6.  7.  8.]]
   [[ 1.  1.  2.  3.  4.]
    [ 1.  1.  2.  3.  4.]
    [ 5.  5.  6.  7.  8.]
    [ 5.  5.  6.  7.  8.]]
   [[ 9.  9. 10. 11. 12.]
    [ 9.  9. 10. 11. 12.]
    [13. 13. 14. 15. 16.]
    [13. 13. 14. 15. 16.]]]]]
```

mindspore.ops.UpsampleTrilinear3D

class mindspore.ops.**UpsampleTrilinear3D**(*align_corners=False*)

Performs upsampling with trilinear interpolation across 3dims for 5dim input Tensor.

This operator scale up the volumetric input with specified *output_size* or *scales* factors, using trilinear upscaling algorithm.

Note: One of *scales* and *output_size* must be specified. And it is an error if both are specified.

Parameters

align_corners (*bool, optional*) –An optional bool. Default: `False`. If `True`, the input and output tensors are aligned by the center points of their corner pixels, preserving the values at the corner pixels. If `False` , the input and output tensors are aligned by the corner points of their corner pixels, and the interpolation use edge value padding for out of boundary values.

Inputs:

- **x** (Tensor) - 5D tensor of shape $(N, C, D_{in}, H_{in}, W_{in})$. Supporting types: [float16, float32, float64].
- **output_size** (Union[tuple[int], list[int]]): A tuple or list of 3 int elements $(output\_depth, output\_height, output\_width)$. Default: `None`.
- **scales** (Union[tuple[float], list[float]]): A tuple or list of 3 float elements $(scale\_depth, scale\_height, scale\_width)$. Default: `None`.

Outputs:

- **y** (Tensor) - Upsampled output with the same data type as *x*, whose shape is $(N, C, D_{out}, H_{out}, W_{out})$.

Raises

- **TypeError** –When *output_size* is not `None` and *output_size* is not list[int] or tuple[int].
- **TypeError** –When *scales* is not `None` and *scales* is not list[float] or tuple[float].
- **TypeError** –If dtype of *x* is not in [float16, float32, float64].
- **TypeError** –If type of *align_corners* is not bool.
- **ValueError** –If any value of *output_size* is negative or zero when *output_size* is not `None`.
- **ValueError** –If any value of *scales* is negative or zero when *scales* is not `None`.
- **ValueError** –If shape of *x* is not 5D.
- **ValueError** –If none of *scales* and *output_size* is specified or both specified.
- **ValueError** –If size of *scales* is not equal 3 when *scales* is specified.
- **ValueError** –If size of *output_size* is not equal 3 when *output_size* is specified.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> net = ops.UpsampleTrilinear3D()
>>> in_x = Tensor(input_data=np.random.randn(2, 3, 4, 512, 256))
>>> output_size=[4, 64, 48]
>>> out = net(in_x, output_size, None)
>>> print(out.shape)
(2, 3, 4, 64, 48)
>>>
>>> net = ops.UpsampleTrilinear3D()
>>> in_x = Tensor(np.arange(1, 5, dtype=np.float32).reshape((1, 1, 1, 2, 2)))
>>> output_size=[2, 4, 4]
>>> out = net(in_x, output_size, None)
>>> print(out)
```

(continues on next page)

(continued from previous page)

```
[[[[1.   1.25 1.75 2.  ]
   [1.5  1.75 2.25 2.5 ]
   [2.5  2.75 3.25 3.5 ]
   [3.   3.25 3.75 4.  ]]
  [[1.   1.25 1.75 2.  ]
   [1.5  1.75 2.25 2.5 ]
   [2.5  2.75 3.25 3.5 ]
   [3.   3.25 3.75 4.  ]]]]
```

18.2.8 Text Processing

API Name	Description	Supported Platforms
mindspore.ops.NoRepeatNGram	Updates the probability of occurrence of words with its corresponding n-grams.	Ascend GPU CPU

mindspore.ops.NoRepeatNGram

class mindspore.ops.**NoRepeatNGram**(*ngram_size=1*)

Updates the probability of occurrence of words with its corresponding n-grams.

During beam search, if consecutive *ngram_size* words exist in the generated word sequence, the consecutive *ngram_size* words will be avoided during subsequent prediction. For example, when *ngram_size* is 3, the generated word sequence is [1, 2, 3, 2, 3], the next predicted word will not be 2 and the value of *log_probs* will be replaced with -FLOAT_MAX. Because 3 consecutive words [2, 3, 2] do not appear twice in the word sequence.

Parameters

ngram_size (*int, optional*) – Size of n-grams, must be greater than 0. Default: `1` .

Inputs:

- **state_seq** (Tensor) - n-gram word series, a 3-D tensor with shape: $(batch\_size, beam\_width, m)$.
- **log_probs** (Tensor) - Probability of occurrence of n-gram word series, a 3-D tensor with shape: $(batch\_size, beam\_width, vocab\_size)$. The value of log_probs will be replaced with -FLOAT_MAX when n-grams repeated.

Outputs:

- **log_probs** (Tensor) - The output Tensor with same shape and type as original *log_probs*.

Raises

- **TypeError** – If *ngram_size* is not an int.
- **TypeError** – If neither *state_seq* nor *log_probs* is a Tensor.
- **TypeError** – If the dtype of *state_seq* is not int.
- **TypeError** – If the dtype of *log_probs* is not float.
- **ValueError** – If *ngram_size* is less than zero.
- **ValueError** – If *ngram_size* is greater than m.
- **ValueError** – If neither *state_seq* nor *log_probs* is not a 3-D Tensor.

- **ValueError** – If the batch_size of *state_seq* and *log_probs* are not equal.
- **ValueError** – If the beam_width of *state_seq* and *log_probs* are not equal.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> no_repeat_ngram = ops.NoRepeatNGram(ngram_size=3)
>>> state_seq = Tensor([[[1, 2, 1, 2, 5, 1, 2],
...                      [9, 3, 9, 5, 4, 1, 5]],
...                     [[4, 8, 6, 4, 5, 6, 4],
...                      [4, 8, 8, 4, 3, 4, 8]]], dtype=mindspore.int32)
>>> log_probs = Tensor([[[0.7, 0.8, 0.6, 0.9, 0.2, 0.8, 0.4, 0.6, 0.2, 0.7],
...                      [0.4, 0.5, 0.6, 0.7, 0.8, 0.1, 0.9, 0.8, 0.7, 0.1]],
...                     [[0.9, 0.7, 0.6, 0.3, 0.5, 0.3, 0.5, 0.4, 0.8, 0.6],
...                      [0.5, 0.8, 0.8, 0.7, 0.7, 0.8, 0.2, 0.7, 0.9, 0.7]]],␣
↪dtype=mindspore.float32)
>>> output = no_repeat_ngram(state_seq, log_probs)
>>> print(output)
[[[ 6.9999999e-01 -3.4028235e+38  6.0000002e-01  8.9999998e-01
    2.0000000e-01 -3.4028235e+38  4.0000001e-01  6.0000002e-01
    2.0000000e-01  6.9999999e-01]
  [ 4.0000001e-01  5.0000000e-01  6.0000002e-01  6.9999999e-01
    8.0000001e-01  1.0000000e-01  8.9999998e-01  8.0000001e-01
    6.9999999e-01  1.0000000e-01]]
 [[ 8.9999998e-01  6.9999999e-01  6.0000002e-01  3.0000001e-01
    5.0000000e-01 -3.4028235e+38  5.0000000e-01  4.0000001e-01
    8.0000001e-01  6.0000002e-01]
  [ 5.0000000e-01  8.0000001e-01  8.0000001e-01  6.9999999e-01
    6.9999999e-01  8.0000001e-01  2.0000000e-01  6.9999999e-01
   -3.4028235e+38  6.9999999e-01]]]
```

18.3 Mathematical Operators

API Name	Description	Supported Platforms
mindspore.ops.Bincount	Counts the number of occurrences of each value in an integer array.	`Ascend` `GPU` `CPU`
mindspore.ops.Cholesky	Performs the Cholesky decomposition on a single or a batch of symmetric positive-definite matrices.	`GPU` `CPU`
mindspore.ops.Complex	Returns a complex Tensor from the real part and the imag part.	`Ascend` `GPU` `CPU`
mindspore.ops.ComplexAbs	Returns a Tensor that contains the magnitudes of the input.	`Ascend` `GPU` `CPU`
mindspore.ops.Cross	Returns the cross product of vectors in dimension *dim* of input and other.	`Ascend` `CPU`
mindspore.ops.FFTWithSize	Fourier transform, can be adjusted by parameters to achieve FFT/IFFT/RFFT/IRFFT.	`Ascend` `GPU` `CPU`

continues on next page

Table 10 – continued from previous page

mindspore.ops.Gcd	Refer to *mindspore.ops.gcd()* for more details.	Ascend GPU CPU

18.3.1 mindspore.ops.Bincount

class mindspore.ops.**Bincount**

Counts the number of occurrences of each value in an integer array.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **array** (Tensor) - A Tensor of type int32, whose value can not be less than zero.
- **size** (Tensor) - A non-negative Tensor of type int32.
- **weights** (Tensor) - A Tensor with the same shape as array, or a length-0 Tensor, in which case it acts as all weights equal to 1. Must be one of the following types: int32, int64, float32, float64.

Outputs:

A Tensor. Has the same type as weights.

Raises

- **TypeError** – If dtype of *array* is not int32.
- **TypeError** – If dtype of *size* is not int32.
- **ValueError** – If *size* is negative.
- **ValueError** – If *weights* are empty.
- **ValueError** – If size of *weights* is not zero and the shape of *weights* is different with the shape of *array*.
- **TypeError** – If dtype of *weights* is not in int32,int64,float32,float64

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> array = Tensor(np.array([1, 2, 2, 3, 3, 3, 4, 4, 4, 4]), mindspore.int32)
>>> size = Tensor(5, mindspore.int32)
>>> weights = Tensor(np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1]), mindspore.float32)
>>> bincount = ops.Bincount()
>>> bins = bincount(array, size, weights)
>>> print(bins)
[0. 1. 2. 3. 4.]
```

18.3.2 mindspore.ops.Cholesky

class mindspore.ops.**Cholesky**(*upper=False*)

Performs the Cholesky decomposition on a single or a batch of symmetric positive-definite matrices.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.cholesky()* for more details.

Parameters

upper (*bool, optional*) – Flag that indicates whether to return a upper or lower triangular matrix. Default: `False` .

Inputs:

- **input_x** (Tensor) - Tensor of shape $(*, N, N)$, where $*$ is zero or more batch dimensions consisting of symmetric positive-definite matrices, with float32 or float64 data type.

Outputs:

Tensor, has the same shape and data type as *input_x*.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[1.0, 1.0], [1.0, 2.0]]), mindspore.float32)
>>> output = ops.Cholesky()(input_x)
>>> print(output)
[[1. 0.]
 [1. 1.]]
```

18.3.3 mindspore.ops.Complex

class mindspore.ops.**Complex**

Returns a complex Tensor from the real part and the imag part.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **real** (Tensor) - The real input tensor. types: float32, float64.
- **imag** (Tensor) - The imag input tensor. types: float32, float64.

Outputs:

Tensor, has the complex type.

Raises

- **TypeError** –If the dtype of input is not one of: float32, float64.
- **TypeError** –If the dtypes of two inputs are not same.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> real = Tensor(np.array([1]), mindspore.float32)
>>> imag = Tensor(np.array([2]), mindspore.float32)
>>> complex = ops.Complex()
>>> output = complex(real, imag)
>>> print(output)
[1.+2.j]
```

18.3.4 mindspore.ops.ComplexAbs

class mindspore.ops.**ComplexAbs**

Returns a Tensor that contains the magnitudes of the input.

The complex numbers in input must be of the form $a + bj$, where a is the real part and b is the imaginary part.

$$y = \sqrt{a^2 + b^2}$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - A Tensor, types: complex64, complex128.

Outputs:

Tensor, has the same shape as x. If the type of x is complex64, the type of output is float32. If the type of x is complex128, the type of output is float64.

Raises

- **TypeError** –If the input is not a Tensor.
- **TypeError** –If the input type is not complex64 or complex128.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.asarray(np.complex(3+4j)), mindspore.complex64)
>>> complex_abs = ops.ComplexAbs()
>>> output = complex_abs(x)
>>> print(output)
5.0
```

18.3.5 mindspore.ops.Cross

class mindspore.ops.**Cross**(*dim=- 65530*)

Returns the cross product of vectors in dimension *dim* of input and other.

Warning: This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.cross()* for more details.

Parameters

dim (*int, optional*) – Specified dim along which to compute cross product with. Default: `-65530`.

Inputs:

- **input** (Tensor) - Input Tensor.
- **other** (Tensor) - Another input Tensor, must have the same shape and the same type as *input*, and the size of their *dim* dimension should be 3.

Outputs:

Tensor, has the same shape and type as inputs.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> from mindspore import ops
>>> cross = ops.Cross(dim = 0)
>>> x1 = Tensor([1, 2, 3], mstype.int8)
>>> x2 = Tensor([1, 2, 3], mstype.int8)
>>> output = cross(x1, x2)
>>> print(output)
[0 0 0]
```

18.3.6 mindspore.ops.FFTWithSize

class mindspore.ops.**FFTWithSize**(*signal_ndim*, *inverse*, *real*, *norm='backward'*, *onesided=True*, *signal_sizes=()*)

Fourier transform, can be adjusted by parameters to achieve FFT/IFFT/RFFT/IRFFT.

For fft, it computes the following expression:

$$X[\omega_1, \ldots, \omega_d] = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} x[n_1, \ldots, n_d] e^{-j\ 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = *signal_ndim* is number of dimensions for the signal, and N_i is the size of signal dimension i.

For ifft, it computes the following expression:

$$X[\omega_1, \ldots, \omega_d] = \frac{1}{\prod_{i=1}^{d} N_i} \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} x[n_1, \ldots, n_d] e^{\ j\ 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = *signal_ndim* is number of dimensions for the signal, and N_i is the size of signal dimension i.

Note:

- FFT/IFFT requires complex64 or complex128 inputs, return complex64 or complex128 outputs.
- RFFT requires bool, uint8, int8, int16, int32, int64, float32 and float64 inputs, return complex64 or complex128 outputs.
- IRFFT requires complex64 or complex128 inputs, return float32 or float64 outputs.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **signal_ndim** (*int*) −The number of dimensions in each signal, this controls how many dimensions of the fourier transform are realized, can only be 1, 2 or 3.
- **inverse** (*bool*) −Whether it is the inverse transformation, used to select from FFT and RFFT or IFFT and IRFFT.
  - when set to `True`: IFFT and IRFFT.
  - when set to `False`: FFT and RFFT.
- **real** (*bool*) −Whether it is the real transformation, combines with *inverse* to select a specific transformation mode:
  - *inverse* is `False` , *real* is `False` : corresponds to FFT.
  - *inverse* is `True` , *real* is `False` : corresponds to IFFT.
  - *inverse* is `False` , *real* is `True` : corresponds to RFFT.
  - *inverse* is `True` , *real* is `True` : corresponds to IRFFT.
- **norm** (*str, optional*) −The normalization, optional values: [`"backward"` , `"forward"` , `"ortho"`]. Default value: `"backward"` .
  - `"backward"` has the direct transforms unscaled and the inverse transforms scaled by $1/n$, where n is the input x's element numbers.
  - `"ortho"` has both direct and inverse transforms are scaled by $1/\sqrt{n}$.
  - `"forward"` has the direct transforms scaled by $1/n$ and the inverse transforms unscaled.

- **onesided** (*bool, optional*) –Controls whether the input is halved to avoid redundancy. Default: `True` .
- **signal_sizes** (*tuple, optional*) –Size of the original signal (the signal before rfft, no batch dimension), only in IRFFT mode and set *onesided* to `True` requires the parameter, the following conditions must be satisfied. Default: `()` .
  - The length of *signal_sizes* is equal to the signal_ndim of the IRFFT: $len(signal\_sizes) = signal\_ndim$.
  - The last dimension of *signal_sizes* divided by 2 is equal to the last dimension of the IRFFT input: $signal\_size[-1]/2 + 1 = x.shape[-1]$.
  - *signal_sizes* has exactly the same dimensions as the input shape except for the last dimension: $signal\_sizes[:-1] = x.shape[:-1]$.

Inputs:

- **x** (Tensor) - The dimension of the input tensor must be greater than or equal to signal_ndim.

Outputs:

A tensor containing the complex-to-complex, real-to-complex or complex-to-real Fourier transform result.

Raises

- **TypeError** –If the input type of FFT/IFFT/IRFFT is not one of: complex64, complex128.
- **TypeError** –If the input type is not Tensor.
- **ValueError** –If *x* dimension is less than signal_ndim.
- **ValueError** –If signal_ndim is greater than 3 or less than 1.
- **ValueError** –If norm is none of "backward", "forward" or "ortho".

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case FFT: signal_ndim: 1, inverse: False, real: False.
>>> fft_in = Tensor(np.array([2, 1, 2]), mindspore.complex64)
>>> fft_net = ops.FFTWithSize(signal_ndim=1, inverse=False, real=False)
>>> fft_output = fft_net(fft_in)
>>> print(fft_output)
[5.        +0.j         0.5       +0.86602545j 0.50000006-0.8660255j ]
>>> # case IFFT: signal_ndim: 1, inverse: True, real: False.
>>> ifft_in = fft_output
>>> ifft_net = ops.FFTWithSize(signal_ndim=1, inverse=True, real=False)
>>> ifft_output = ifft_net(ifft_in)
>>> print(ifft_output)
[2.        -1.9868216e-08j 0.99999994+0.0000000e+00j
 1.9999999 +7.9472862e-08j]
>>> # case RFFT2D: signal_ndim: 2, inverse: False, real: True.
>>> rfft_in = Tensor(np.array([[2, 1, 2], [3, 1, 6]]), mindspore.float32)
>>> rfft_net = ops.FFTWithSize(signal_ndim=2, inverse=False, real=True)
>>> rfft_output = rfft_net(rfft_in)
```

(continues on next page)

(continued from previous page)

```
>>> print(rfft_output)
[[ 1.5000000e+01+1.1920929e-07j -2.3841858e-07+5.1961522e+00j]
 [-5.0000000e+00-2.9802322e-08j  9.9999988e-01-3.4641016e+00j]]
>>> # case IRFFT2D: signal_ndim: 2, inverse: True, real: True.
>>> irfft_in = rfft_output
>>> irfft_net = ops.FFTWithSize(signal_ndim=2, inverse=True, real=True, signal_sizes=rfft_in.
↪shape)
>>> irfft_output = irfft_net(irfft_in)
>>> print(irfft_output)
[[2.         1.         2.        ]
 [3.         0.99999994 5.9999995 ]]
```

18.3.7 mindspore.ops.Gcd

class mindspore.ops.**Gcd**

```
prim = ops.Gcd()
out = prim(input, other)
```

is equivalent to

```
ops.gcd(input, other)
```

Refer to *mindspore.ops.gcd()* for more details.

18.3.8 Element-wise Operator

API Name	Description	Supported Platforms
mindspore.ops.Abs	Refer to *mindspore.ops.abs()* for more details.	Ascend GPU CPU
mindspore.ops.AccumulateNV2	Computes accumulation of all input tensors element-wise.	Ascend GPU
mindspore.ops.ACos	Refer to *mindspore.ops.acos()* for more details.	Ascend GPU CPU
mindspore.ops.Acosh	Refer to *mindspore.ops.acosh()* for more details.	Ascend GPU CPU
mindspore.ops.Add	Refer to *mindspore.ops.add()* for more details.	Ascend GPU CPU
mindspore.ops.Addcdiv	Adds the element-wise division of *x1* by *x2*, multiplied by *value* to *input_data*.	Ascend GPU CPU
mindspore.ops.Addcmul	Adds the element-wise product of *x1* by *x2*, multiplied by *value* to *input_data*.	Ascend GPU CPU
mindspore.ops.AddN	Computes addition of all input tensors element-wise.	Ascend GPU CPU
mindspore.ops.Angle	Refer to *mindspore.ops.angle()* for more details.	Ascend GPU CPU
mindspore.ops.Asin	Refer to *mindspore.ops.asin()* for more details.	Ascend GPU CPU
mindspore.ops.Asinh	Refer to *mindspore.ops.asinh()* for more details.	Ascend GPU CPU

continues on next page

Table 11 – continued from previous page

mindspore.ops.Atan	Refer to mindspore.ops.atan() for more details.	Ascend GPU CPU
mindspore.ops.Atan2	Refer to mindspore.ops.atan2() for more details.	Ascend GPU CPU
mindspore.ops.Atanh	Refer to mindspore.ops.atanh() for more details.	Ascend GPU CPU
mindspore.ops.BesselI0	Computes modified Bessel function of the first kind, order 0 element-wise.	GPU CPU
mindspore.ops.BesselI0e	Computes exponential scaled modified Bessel function of the first kind, order 0 element-wise.	Ascend GPU CPU
mindspore.ops.BesselI1	Computes modified Bessel function of the first kind, order 1 element-wise.	GPU CPU
mindspore.ops.BesselI1e	Computes exponential scaled modified Bessel function of the first kind, order 1 element-wise.	Ascend GPU CPU
mindspore.ops.BesselJ0	Computes Bessel function of the first kind, order 0 element-wise.	GPU CPU
mindspore.ops.BesselJ1	Computes Bessel function of the first kind, order 1 element-wise.	GPU CPU
mindspore.ops.BesselK0	Computes modified Bessel function of the second kind, order 0 element-wise.	GPU CPU
mindspore.ops.BesselK0e	Computes exponential scaled modified Bessel function of the second kind, order 0 element-wise.	GPU CPU
mindspore.ops.BesselK1	Computes modified Bessel function of the second kind, order 1 element-wise.	GPU CPU
mindspore.ops.BesselK1e	Computes exponential scaled modified Bessel function of the second kind, order 1 element-wise.	GPU CPU
mindspore.ops.BesselY0	Computes Bessel function of the second kind, order 0 element-wise.	GPU CPU
mindspore.ops.BesselY1	Computes Bessel function of the second kind, order 1 element-wise.	GPU CPU
mindspore.ops.BitwiseAnd	Returns bitwise *and* of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.BitwiseOr	Returns bitwise *or* of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.BitwiseXor	Returns bitwise *xor* of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.Ceil	Refer to mindspore.ops.ceil() for more details.	Ascend GPU CPU
mindspore.ops.Conj	Refer to mindspore.ops.conj() for more details.	Ascend GPU CPU
mindspore.ops.Cos	Refer to mindspore.ops.cos() for more details.	Ascend GPU CPU
mindspore.ops.Cosh	Refer to mindspore.ops.cosh() for more details.	Ascend GPU CPU
mindspore.ops.Digamma	Computes the grad of the lgamma function on input.	GPU CPU
mindspore.ops.Div	Computes the quotient of dividing the first input tensor by the second input tensor element-wise.	Ascend GPU CPU
mindspore.ops.DivNoNan	Operates a safe division between *x1* and *x2* element-wise.	Ascend GPU CPU
mindspore.ops.Einsum	Sums the product of the elements of the input Tensor along dimensions specified notation based on the Einstein summation convention(Einsum).	GPU

continues on next page

Table 11 – continued from previous page

mindspore.ops.Erf	Refer to mindspore.ops.erf() for more details.	Ascend GPU CPU
mindspore.ops.Erfc	Refer to mindspore.ops.erfc() for more details.	Ascend GPU CPU
mindspore.ops.Erfinv	Refer to mindspore.ops.erfinv() for more details.	Ascend GPU CPU
mindspore.ops.Exp	Refer to mindspore.ops.exp() for more details.	Ascend GPU CPU
mindspore.ops.Expm1	Refer to mindspore.ops.expm1() for more details.	Ascend GPU CPU
mindspore.ops.Floor	Refer to mindspore.ops.floor() for more details.	Ascend GPU CPU
mindspore.ops.FloorDiv	Refer to mindspore.ops.floor_divide() for more details.	Ascend GPU CPU
mindspore.ops.FloorMod	Refer to mindspore.ops.floor_mod() for more details.	Ascend GPU CPU
mindspore.ops.Geqrf	Refer to mindspore.ops.geqrf() for more details.	Ascend GPU CPU
mindspore.ops.Imag	Returns a new tensor containing imaginary value of the input.	Ascend GPU CPU
mindspore.ops.Inv	Computes Reciprocal of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Invert	Flips all bits of input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Lerp	Refer to mindspore.ops.lerp() for more details.	Ascend GPU CPU
mindspore.ops.Log	Refer to mindspore.ops.log() for more details.	Ascend GPU CPU
mindspore.ops.Log1p	Refer to mindspore.ops.log1p() for more details.	Ascend GPU CPU
mindspore.ops.LogicalAnd	Computes the "logical AND" of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.LogicalNot	Computes the "logical NOT" of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.LogicalOr	Computes the "logical OR" of two tensors element-wise.	Ascend GPU CPU
mindspore.ops.LogicalXor	Computes the "logical XOR" of two tensors element-wise.	Ascend CPU
mindspore.ops.Logit	Calculate the logit of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.Mod	Computes the remainder of dividing the first input tensor by the second input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Mul	Refer to mindspore.ops.mul() for more details.	Ascend GPU CPU
mindspore.ops.MulNoNan	Computes $x * y$ element-wise.	Ascend GPU CPU
mindspore.ops.Neg	Refer to mindspore.ops.neg() for more details.	Ascend GPU CPU
mindspore.ops.NextAfter	Refer to mindspore.ops.nextafter() for more details.	Ascend GPU CPU
mindspore.ops.Pow	Refer to mindspore.ops.pow() for more details.	Ascend GPU CPU
mindspore.ops.Polar	Converts polar coordinates to Cartesian coordinates.	Ascend GPU CPU
mindspore.ops.Polygamma	Computes the a.	GPU CPU

continues on next page

Table 11 – continued from previous page

mindspore.ops.Real	Refer to mindspore.ops.real() for more details.	Ascend GPU CPU
mindspore.ops.RealDiv	Divides the first input tensor by the second input tensor in floating-point type element-wise.	Ascend GPU CPU
mindspore.ops.Reciprocal	Returns reciprocal of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.Rint	Returns an integer that is closest to *input_x* element-wise.	Ascend GPU CPU
mindspore.ops.Round	Returns half to even of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.Rsqrt	Refer to mindspore.ops.rsqrt() for more details.	Ascend GPU CPU
mindspore.ops.Sign	Refer to mindspore.ops.sign() for more details.	Ascend GPU CPU
mindspore.ops.Sin	Refer to mindspore.ops.sin() for more details.	Ascend GPU CPU
mindspore.ops.Sinc	Refer to mindspore.ops.sinc() for more details.	Ascend GPU CPU
mindspore.ops.Sinh	Refer to mindspore.ops.sinh() for more details.	Ascend GPU CPU
mindspore.ops.Sqrt	Refer to mindspore.ops.sqrt() for more details.	Ascend GPU CPU
mindspore.ops.Square	Refer to mindspore.ops.square() for more details.	Ascend GPU CPU
mindspore.ops.SquaredDifference	Subtracts the second input tensor from the first input tensor element-wise and returns square of it.	Ascend GPU CPU
mindspore.ops.SquareSumAll	Returns the square sum of a tensor element-wise.	Ascend GPU CPU
mindspore.ops.Sub	Subtracts the second input tensor from the first input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Tan	Computes tangent of *input* element-wise.	Ascend GPU CPU
mindspore.ops.Trunc	Refer to mindspore.ops.trunc() for more details.	Ascend GPU CPU
mindspore.ops.TruncateDiv	Divides the first input tensor by the second input tensor element-wise and rounds the results of division towards zero.	Ascend GPU CPU
mindspore.ops.TruncateMod	Returns the remainder of division element-wise.	Ascend GPU CPU
mindspore.ops.Xdivy	Divides the first input tensor by the second input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Xlogy	Computes the first input tensor multiplied by the logarithm of second input tensor element-wise.	Ascend GPU CPU
mindspore.ops.Zeta	Compute the Hurwitz zeta function ζ(x,q) of input Tensor.	Ascend GPU CPU

mindspore.ops.Abs

class mindspore.ops.**Abs**

```
prim = ops.Abs()
out = prim(input)
```

is equivalent to

```
ops.abs(input)
```

Refer to *mindspore.ops.abs()* for more details.

mindspore.ops.AccumulateNV2

class mindspore.ops.**AccumulateNV2**

Computes accumulation of all input tensors element-wise.

Refer to *mindspore.ops.accumulate_n()* for more details.

Inputs:

- **x** (Union(tuple[Tensor], list[Tensor])) - The input tuple or list is made up of multiple tensors whose dtype is number to be added together. Each element of tuple or list should have the same shape.

Outputs:

Tensor, has the same shape and dtype as each entry of the *x*.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, nn
>>> class NetAccumulateNV2(nn.Cell):
...     def __init__(self):
...         super(NetAccumulateNV2, self).__init__()
...         self.accumulateNV2 = ops.AccumulateNV2()
...
...     def construct(self, *z):
...         return self.accumulateNV2(z)
...
>>> net = NetAccumulateNV2()
>>> x = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> y = Tensor(np.array([4, 5, 6]), mindspore.float32)
>>> output = net(x, y, x, y)
>>> print(output)
[10. 14. 18.]
```

mindspore.ops.ACos

class mindspore.ops.**ACos**

```
prim = ops.ACos()
out = prim(input)
```

is equivalent to

```
ops.acos(input)
```

Refer to *mindspore.ops.acos()* for more details.

mindspore.ops.Acosh

class mindspore.ops.**Acosh**

```
prim = ops.Acosh()
out = prim(input)
```

is equivalent to

```
ops.acosh(input)
```

Refer to *mindspore.ops.acosh()* for more details.

mindspore.ops.Add

class mindspore.ops.**Add**

```
prim = ops.Add()
out = prim(input, other)
```

is equivalent to

```
ops.add(input, other)
```

Refer to *mindspore.ops.add()* for more details.

mindspore.ops.Addcdiv

class mindspore.ops.**Addcdiv**

Adds the element-wise division of *x1* by *x2*, multiplied by *value* to *input_data*. It computes the following operation:

$$y[i] = input\_data[i] + value[i] * (x1[i]/x2[i])$$

Inputs:

- **input_data** (Tensor) - The tensor to be added.
- **x1** (Tensor) - The numerator tensor.
- **x2** (Tensor) - The denominator tensor.
- **value** (Tensor) - The multiplier for tensor x1/x2.

Outputs:

Tensor, has the same shape and dtype as x1/x2.

Raises

- **TypeError** –If dtype of *x1*, *x2*, *value*, *input_data* is not tensor.
- **TypeError** –If dtype of *x1*, *x2*, *value*, *input_data* are not the same.
- **ValueError** –If *x1* could not be broadcast to *x2*.
- **ValueError** –If *value* could not be broadcast to *x1/x2*.
- **ValueError** –If *input_data* could not be broadcast to *value*(x1/x2)*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_data = Tensor(np.array([1, 1, 1, 1]), mindspore.float32)
>>> x1 = Tensor(np.array([1, 2, 3, 4]), mindspore.float32)
>>> x2 = Tensor(np.array([4, 3, 2, 1]), mindspore.float32)
>>> value = Tensor([1], mindspore.float32)
>>> addcdiv = ops.Addcdiv()
>>> y = addcdiv(input_data, x1, x2, value)
>>> print(y)
[1.25      1.6666667 2.5       5.       ]
```

mindspore.ops.Addcmul

class mindspore.ops.**Addcmul**

Adds the element-wise product of *x1* by *x2*, multiplied by *value* to *input_data*. It computes the following operation:

$$output[i] = input\_data[i] + value[i] * (x1[i] * x2[i])$$

Inputs:

- **input_data** (Tensor) - The tensor to be added.
- **x1** (Tensor) - The tensor to be multiplied.
- **x2** (Tensor) - The tensor to be multiplied.
- **value** (Tensor) - The multiplier for tensor x1*x2.

Outputs:

Tensor, has the same shape and dtype as x1*x2.

Raises

- **TypeError** – If dtype of *x1*, *x2*, *value*, *input_data* is not tensor.
- **TypeError** – If dtype of *x1*, *x2*, *value*, *input_data* are not the same.
- **ValueError** – If *x1* could not be broadcast to *x2*.
- **ValueError** – If *value* could not be broadcast to *x1 * x2*.
- **ValueError** – If *input_data* could not be broadcast to *value*(x1*x2)*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_data = Tensor(np.array([1, 1, 1]), mindspore.float32)
>>> x1 = Tensor(np.array([[1], [2], [3]]), mindspore.float32)
>>> x2 = Tensor(np.array([[1, 2, 3]]), mindspore.float32)
>>> value = Tensor([1], mindspore.float32)
>>> addcmul = ops.Addcmul()
>>> y = addcmul(input_data, x1, x2, value)
>>> print(y)
[[ 2.  3.  4.]
 [ 3.  5.  7.]
 [ 4.  7. 10.]]
```

mindspore.ops.AddN

class mindspore.ops.**AddN**

Computes addition of all input tensors element-wise.

Refer to *mindspore.ops.addn()* for more details.

Inputs:

- **x** (Union(tuple[Tensor], list[Tensor])) - A tuple or list composed of Tensor, the data type is boolean or numeric.

Outputs:

Tensor, has the same shape and dtype as each Tensor of *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> class NetAddN(nn.Cell):
...     def __init__(self):
...         super(NetAddN, self).__init__()
...         self.addN = ops.AddN()
...
...     def construct(self, *z):
...         return self.addN(z)
...
>>> net = NetAddN()
>>> x = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> y = Tensor(np.array([4, 5, 6]), mindspore.float32)
>>> output = net(x, y, x, y)
>>> print(output)
[10. 14. 18.]
```

mindspore.ops.Angle

class mindspore.ops.**Angle**

```
prim = ops.Angle()
out = prim(input)
```

is equivalent to

```
ops.angle(input)
```

Refer to *mindspore.ops.angle()* for more details.

mindspore.ops.Asin

class mindspore.ops.**Asin**

```
prim = ops.Asin()
out = prim(input)
```

is equivalent to

```
ops.asin(input)
```

Refer to *mindspore.ops.asin()* for more details.

mindspore.ops.Asinh

class mindspore.ops.**Asinh**

```
prim = ops.Asinh()
out = prim(input)
```

is equivalent to

```
ops.asinh(input)
```

Refer to *mindspore.ops.asinh()* for more details.

mindspore.ops.Atan

class mindspore.ops.**Atan**

```
prim = ops.Atan()
out = prim(input)
```

is equivalent to

```
ops.atan(input)
```

Refer to *mindspore.ops.atan()* for more details.

mindspore.ops.Atan2

class mindspore.ops.**Atan2**

```
prim = ops.Atan2()
out = prim(input, other)
```

is equivalent to

```
ops.atan2(input, other)
```

Refer to *mindspore.ops.atan2()* for more details.

mindspore.ops.Atanh

class mindspore.ops.**Atanh**

```
prim = ops.Atanh()
out = prim(input)
```

is equivalent to

```
ops.atanh(input)
```

Refer to *mindspore.ops.atanh()* for more details.

mindspore.ops.BesselI0

class mindspore.ops.**BesselI0**

Computes modified Bessel function of the first kind, order 0 element-wise.

The formula is defined as:

$$I_0(x) = J_0(\mathrm{i}x) = \sum_{m=0}^{\infty} \frac{x^{2m}}{2^{2m}(m!)^2}$$

where J_0 is Bessel function of the first kind, order 0.

Warning: This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.bessel_i0()* for more details.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as x.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_i0 = ops.BesselI0()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_i0(x)
>>> print(output)
[1.0144521 1.1797839 1.0241698 1.0020262]
```

mindspore.ops.BesselI0e

class mindspore.ops.**BesselI0e**

Computes exponential scaled modified Bessel function of the first kind, order 0 element-wise.

The formula is defined as:

$$\mathrm{I}_0e(x) = e^{(-|x|)} * I_0(x) = e^{(-|x|)} * \sum_{m=0}^{\infty} \frac{x^{2m}}{2^{2m}(m!)^2}$$

where I_0 is modified Bessel function of the first kind, order 0.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as x.

Raises

- **TypeError** – If x is not a Tensor.
- **TypeError** – If dtype of x is not float16, float32 or float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_i0e = ops.BesselI0e()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_i0e(x)
>>> print(output)
[0.7979961  0.5144438  0.75117415  0.9157829 ]
```

mindspore.ops.BesselI1

class mindspore.ops.**BesselI1**

Computes modified Bessel function of the first kind, order 1 element-wise.

The formula is defined as:

$$I_1(x) = \mathrm{i}^{-1} J_1(\mathrm{i}x) = \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2^{2m+1} m!(m+1)!}$$

where J_1 is Bessel function of the first kind, order 1.

Warning: This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.bessel_i1()* for more details.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as x.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_i1 = ops.BesselI1()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_i1(x)
>>> print(output)
[0.1208661  0.45177728 0.1568694  0.04504559]
```

mindspore.ops.BesselI1e

class mindspore.ops.**BesselI1e**

Computes exponential scaled modified Bessel function of the first kind, order 1 element-wise.

The formula is defined as:

$$\mathrm{I}_1 e(x) = e^{(-|x|)} * I_1(x) = e^{(-|x|)} * \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2^{2m+1} m!(m+1)!}$$

where I_1 is modified Bessel function of the first kind, order 1.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16 or float32, float64.

Outputs:

Tensor, has the same shape as x.

Raises

- **TypeError** – If x is not a Tensor.
- **TypeError** – If dtype of x is not float16, float32 or float64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_i1e = ops.BesselI1e()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_i1e(x)
>>> print(output)
[0.09507662 0.19699717 0.11505538 0.04116856]
```

mindspore.ops.BesselJ0

class mindspore.ops.**BesselJ0**

Computes Bessel function of the first kind, order 0 element-wise.

The formula is defined as:

$$J_0(x) = \frac{1}{\pi}\int_0^{\pi}\cos(x\sin\theta)d\theta = \sum_{m=0}^{\infty}\frac{(-1)^m x^{2m}}{2^{2m}(m!)^2}$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as x.

Raises

TypeError – If x is not a Tensor of float16, float32 or float64.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_j0 = ops.BesselJ0()
>>> x = Tensor(np.array([0.5, 1., 2., 4.]), mindspore.float32)
>>> output = bessel_j0(x)
>>> print(output)
[0.93846981  0.76519769  0.22389078  -0.39714981]
```

mindspore.ops.BesselJ1

class mindspore.ops.**BesselJ1**

Computes Bessel function of the first kind, order 1 element-wise.

The formula is defined as:

$$J_1(x) = \frac{1}{\pi}\int_0^{\pi}\cos(x\sin\theta-\theta)d\theta = \sum_{m=0}^{\infty}\frac{(-1)^m x^{2m+1}}{2^{2m+1}m!(m+1)!}$$

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32 or float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_j1 = ops.BesselJ1()
>>> x = Tensor(np.array([0.5, 1., 2., 4.]), mindspore.float32)
>>> output = bessel_j1(x)
>>> print(output)
[0.24226846  0.44005059  0.57672481  -0.06604333]
```

mindspore.ops.BesselK0

class mindspore.ops.**BesselK0**

Computes modified Bessel function of the second kind, order 0 element-wise.

The formula is defined as:

$$K_0(x) = \lim_{\nu \to 0} \left(\frac{\pi}{2}\right) \frac{I_{-\nu}(x) - I_{\nu}(x)}{\sin(\nu\pi)} = \int_0^{\infty} e^{-x\cosh t} dt$$

where I_0 is modified Bessel function of the first kind, order 0.

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32, float64.

Outputs:

Tensor, has the same shape as x.

Raises

TypeError – If x is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_k0 = ops.BesselK0()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_k0(x)
>>> print(output)
[1.579826  0.5402144 1.3424659 2.5310173]
```

mindspore.ops.BesselK0e

class mindspore.ops.**BesselK0e**

Computes exponential scaled modified Bessel function of the second kind, order 0 element-wise.

The formula is defined as:

$$K_0e(x) = e^{(-|x|)} * K_0(x) = e^{(-|x|)} * \int_0^{\infty} e^{-x\cosh t} dt$$

where K_0 is modified Bessel function of the second kind, order 0.

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32, float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_k0e = ops.BesselK0e()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_k0e(x)
>>> print(output)
[2.0083523 1.2388839 1.8303517 2.769374 ]
```

mindspore.ops.BesselK1

class mindspore.ops.**BesselK1**

Computes modified Bessel function of the second kind, order 1 element-wise.

The formula is defined as:

$$K_1(x) = \lim_{\nu \to 1} \left(\frac{\pi}{2}\right) \frac{I_{-\nu}(x) - I_{\nu}(x)}{\sin(\nu\pi)} = \int_0^{\infty} e^{-x\cosh t} \cosh(t) dt$$

where I_1 is modified Bessel function of the first kind, order 1.

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32, float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_k1 = ops.BesselK1()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_k1(x)
>>> print(output)
[3.9190812  0.8143549  2.9440577 10.974864]
```

mindspore.ops.BesselK1e

class mindspore.ops.**BesselK1e**

Computes exponential scaled modified Bessel function of the second kind, order 1 element-wise.

The formula is defined as:

$$K_1e(x) = e^{(-|x|)} * K_1(x) = e^{(-|x|)} * \int_0^\infty e^{-x\cosh t}\cosh(t)dt$$

where K_1 is modified Bessel function of the second kind, order 1.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32, float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_k1e = ops.BesselK1e()
>>> x = Tensor(np.array([0.24, 0.83, 0.31, 0.09]), mindspore.float32)
>>> output = bessel_k1e(x)
>>> print(output)
[ 4.9821286  1.8675754  4.0140023 12.008413 ]
```

mindspore.ops.BesselY0

class mindspore.ops.**BesselY0**

Computes Bessel function of the second kind, order 0 element-wise.

The formula is defined as:

$$Y_0(x) = \lim_{n \to 0} \frac{J_n(x)\cos n\pi - J_{-n}(x)}{\sin n\pi}$$

where J_0 is Bessel function of the first kind, order 0.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_y0 = ops.BesselY0()
>>> x = Tensor(np.array([0.5, 1., 2., 4.]), mindspore.float32)
>>> output = bessel_y0(x)
>>> print(output)
[-0.44451873  0.08825696  0.51037567  -0.01694074]
```

mindspore.ops.BesselY1

class mindspore.ops.**BesselY1**

Computes Bessel function of the second kind, order 1 element-wise.

The formula is defined as:

$$Y_1(x) = \lim_{n \to 1} \frac{J_n(x)\cos n\pi - J_{-n}(x)}{\sin n\pi}$$

where J_1 is Bessel function of the first kind, order 1.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. Data type must be float16, float32 or float64.

Outputs:

Tensor, has the same shape as *x*.

Raises

TypeError – If *x* is not a Tensor of float16, float32, float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> bessel_y1 = ops.BesselY1()
>>> x = Tensor(np.array([0.5, 1., 2., 4.]), mindspore.float32)
>>> output = bessel_y1(x)
>>> print(output)
[-1.47147239  -0.78121282  -0.10703243  0.39792571]
```

mindspore.ops.BitwiseAnd

class mindspore.ops.**BitwiseAnd**

Returns bitwise *and* of two tensors element-wise.

Refer to *mindspore.ops.bitwise_and()* for more details.

Inputs:

- **x** (Tensor) - The first input tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **y** (Tensor) - The second input tensor with same type as the *x*.

Outputs:

Tensor, has the same type as the *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> y = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> bitwise_and = ops.BitwiseAnd()
>>> output = bitwise_and(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[ 0  0  1 -1  1  0  1]
```

mindspore.ops.BitwiseOr

class mindspore.ops.**BitwiseOr**

Returns bitwise *or* of two tensors element-wise.

Refer to *mindspore.ops.bitwise_or()* for more details.

Inputs:

- **x** (Tensor) - The first input tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **y** (Tensor) - The second input tensor with same type as the *x*.

Outputs:

Tensor, has the same type as the *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> y = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> bitwise_or = ops.BitwiseOr()
>>> output = bitwise_or(x, y)
>>> print(output)
[ 0  1  1 -1 -1  3  3]
```

mindspore.ops.BitwiseXor

class mindspore.ops.**BitwiseXor**

Returns bitwise *xor* of two tensors element-wise.

> **Warning:** This API has poor performance on CPU and it is recommended to run it on the Ascend/GPU.

Refer to *mindspore.ops.bitwise_xor()* for more details.

Inputs:

- **x** (Tensor) - The first input tensor with shape $(N, *)$ where $*$ means, any number of additional dimensions.
- **y** (Tensor) - The second input tensor with same type as the *x*.

Outputs:

Tensor, has the same type as the *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0, 0, 1, -1, 1, 1, 1]), mindspore.int16)
>>> y = Tensor(np.array([0, 1, 1, -1, -1, 2, 3]), mindspore.int16)
>>> bitwise_xor = ops.BitwiseXor()
>>> output = bitwise_xor(x, y)
>>> print(output)
[ 0  1  0  0 -2  3  2]
```

mindspore.ops.Ceil

class mindspore.ops.**Ceil**

```
prim = ops.Ceil()
out = prim(input)
```

is equivalent to

```
ops.ceil(input)
```

Refer to *mindspore.ops.ceil()* for more details.

mindspore.ops.Conj

class mindspore.ops.**Conj**

```
prim = ops.Conj()
out = prim(input)
```

is equivalent to

```
ops.conj(input)
```

Refer to *mindspore.ops.conj()* for more details.

mindspore.ops.Cos

class mindspore.ops.**Cos**

```
prim = ops.Cos()
out = prim(input)
```

is equivalent to

```
ops.cos(input)
```

Refer to *mindspore.ops.cos()* for more details.

mindspore.ops.Cosh

class mindspore.ops.**Cosh**

```
prim = ops.Cosh()
out = prim(input)
```

is equivalent to

```
ops.cosh(input)
```

Refer to *mindspore.ops.cosh()* for more details.

mindspore.ops.Digamma

class mindspore.ops.**Digamma**

Computes the grad of the lgamma function on input.

$$P(x) = grad(ln(gamma(x)))$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. With type of float16 or float32 or float64.

Outputs:

Tensor, has the same dtype as *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If dtype of input *x* is not float16 or float32 or float64.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.5, 0.5, 9]).astype(np.float16))
>>> digamma = ops.Digamma()
>>> output = digamma(x)
>>> print(output)
[ 0.0365 -1.964   2.14  ]
```

mindspore.ops.Div

class mindspore.ops.**Div**

Computes the quotient of dividing the first input tensor by the second input tensor element-wise.

Refer to *mindspore.ops.div()* for more details.

Note:

- One of the two inputs must be a Tensor, when the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- The two inputs comply with the implicit type conversion rules to make the data types consistent.

Inputs:

- **x** (Union[Tensor, number.Number, bool]) - The first input is a number.Number or a bool or a tensor whose data type is number or bool_.
- **y** (Union[Tensor, number.Number, bool]) - The second input is a number.Number or a bool or a tensor whose data type is number or bool_.

Outputs:

Tensor, the shape is the same as the one of the input x , y after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1 :has same data type and shape of the two inputs
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]), mindspore.float32)
>>> y = Tensor(np.array([3.0, 2.0, 3.0]), mindspore.float32)
>>> div = ops.Div()
>>> output = div(x, y)
>>> print(output)
[-1.3333334  2.5        2.        ]
>>> # case 2 : different data type and shape of the two inputs
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]), mindspore.float32)
>>> y = Tensor(2, mindspore.int32)
>>> output = div(x, y)
>>> print(output)
[-2.  2.5  3.]
>>> print(output.dtype)
Float32
```

mindspore.ops.DivNoNan

class mindspore.ops.**DivNoNan**

Operates a safe division between *x1* and *x2* element-wise. Returns 0 if element of *x2* is zero.

Inputs of *x1* and *x2* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

$$output_i = \begin{cases} 0, & \text{if } x2_i = 0 \\ x1_i / x2_i, & \text{if } x2_i \neq 0 \end{cases}$$

Inputs:

- **x1** (Union[Tensor, number.Number, bool]) - The first input is a number.Number or a bool or a tensor whose data type is number or bool_.
- **x2** (Union[Tensor, number.Number, bool]) - The second input is a number.Number or a bool when the first input is a bool or a tensor whose data type is number or bool_. When the first input is Scalar, the second input must be a Tensor whose data type is number or bool_.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

TypeError – If *x1* and *x2* is not a number.Number or a bool or a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([-1.0, 0., 1.0, 5.0, 6.0]), mindspore.float32)
>>> x2 = Tensor(np.array([0., 0., 0., 2.0, 3.0]), mindspore.float32)
>>> div_no_nan = ops.DivNoNan()
>>> output = div_no_nan(x1, x2)
>>> print(output)
[0.  0.  0.  2.5 2. ]
```

mindspore.ops.Einsum

class mindspore.ops.**Einsum**(*equation*)

Sums the product of the elements of the input Tensor along dimensions specified notation based on the Einstein summation convention(Einsum). You can use this operator to perform diagonal/reducesum/transpose/matmul/mul/inner product operations, etc.

Parameters

equation (*str*) – An attribute, represent the operation you want to do. the value can contain only letters([a-z][A-Z]), commas(,), ellipsis(···), and arrow(->). the letters represent inputs's tensor dimension, commas(,)represent separate tensors, ellipsis(···) indicates the tensor dimension that you do not care about, the left of the arrow(->) indicates the input tensors, and the right of it indicates the desired output dimension.

Inputs:

- **x** () - Input tensor used for calculation. The inputs must be a tuple/list of Tensors. When the inputs are only one tensor, you can input (tensor,). Dtypes of them should be float16/float32/float64 and dtype of the tensor(s) must be the same.

Outputs:

Tensor, the shape of it can be obtained from the equation, and the data type is the same as input tensors.

Raises

- **TypeError** – If equation itself is invalid, or the equation does not match the input tensor.
- **TypeError** – If dtype of the input Tensors are not the same or dtype is not float16, float32 or float64.

Supported Platforms:

GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> equation = "i->"
>>> einsum = ops.Einsum(equation)
>>> output = einsum([x])
>>> print(output)
[7.]
>>>
>>> x = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> y = Tensor(np.array([2.0, 4.0, 3.0]), mindspore.float32)
>>> equation = "i,i->i"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x, y))
>>> print(output)
[ 2. 8. 12.]
>>>
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> y = Tensor(np.array([[2.0, 3.0], [1.0, 2.0], [4.0, 5.0]]), mindspore.float32)
>>> equation = "ij,jk->ik"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x, y))
>>> print(output)
[[16. 22.]
[37. 52.]]
>>>
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "ij->ji"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x,))
>>> print(output)
```

(continues on next page)

(continued from previous page)

```
[[1. 4.]
[2. 5.]
[3. 6.]]
>>>
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "ij->j"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x,))
>>> print(output)
[5. 7. 9.]
>>>
>>> x = Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.float32)
>>> equation = "...->"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x,))
>>> print(output)
[21.]
>>>
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([2.0, 4.0, 1.0]), mindspore.float32)
>>> equation = "j,i->ji"
>>> einsum = ops.Einsum(equation)
>>> output = einsum((x, y))
>>> print(output)
[[ 2. 4. 1.]
[ 4. 8. 2.]
[ 6. 12. 3.]]
```

mindspore.ops.Erf

class mindspore.ops.**Erf**

```
prim = ops.Erf()
out = prim(input)
```

is equivalent to

```
ops.erf(input)
```

Refer to *mindspore.ops.erf()* for more details.

mindspore.ops.Erfc

class mindspore.ops.**Erfc**

```
prim = ops.Erfc()
out = prim(input)
```

is equivalent to

```
ops.erfc(input)
```

Refer to *mindspore.ops.erfc()* for more details.

mindspore.ops.Erfinv

class mindspore.ops.**Erfinv**

```
prim = ops.Erfinv()
out = prim(input)
```

is equivalent to

```
ops.erfinv(input)
```

Refer to *mindspore.ops.erfinv()* for more details.

mindspore.ops.Exp

class mindspore.ops.**Exp**

```
prim = ops.Exp()
out = prim(input)
```

is equivalent to

```
ops.exp(input)
```

Refer to *mindspore.ops.exp()* for more details.

mindspore.ops.Expm1

class mindspore.ops.**Expm1**

```
prim = ops.Expm1()
out = prim(input)
```

is equivalent to

```
ops.expm1(input)
```

Refer to *mindspore.ops.expm1()* for more details.

mindspore.ops.Floor

class mindspore.ops.**Floor**

```
prim = ops.Floor()
out = prim(input)
```

is equivalent to

```
ops.floor(input)
```

Refer to *mindspore.ops.floor()* for more details.

mindspore.ops.FloorDiv

class mindspore.ops.**FloorDiv**

```
prim = ops.FloorDiv()
out = prim(input, other)
```

is equivalent to

```
ops.floor_divide(input, other)
```

Refer to *mindspore.ops.floor_divide()* for more details.

mindspore.ops.FloorMod

class mindspore.ops.**FloorMod**

```
prim = ops.FloorMod()
out = prim(x, y)
```

is equivalent to

```
ops.floor_mod(x, y)
```

Refer to *mindspore.ops.floor_mod()* for more details.

mindspore.ops.Geqrf

class mindspore.ops.**Geqrf**

```
prim = ops.Geqrf()
out = prim(input)
```

is equivalent to

```
ops.geqrf(input)
```

Refer to *mindspore.ops.geqrf()* for more details.

mindspore.ops.Imag

class mindspore.ops.**Imag**

Returns a new tensor containing imaginary value of the input. If input is real, it is returned zeros.

Inputs:

- **input** (Tensor) - The input tensor.

Outputs:

Tensor, the shape is the same as the input.

Raises

TypeError – If the input is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> x = mindspore.tensor(1.3+0.4j, mindspore.complex64)
>>> imag = ops.Imag()
>>> output = imag(x)
>>> print(output)
0.4
```

mindspore.ops.Inv

class mindspore.ops.**Inv**

Computes Reciprocal of input tensor element-wise.

Refer to *mindspore.ops.inv()* for more details.

Inputs:

- **x** (Tensor) - Input tensor, it must be one of the following types: float16, float32 or int32.

Outputs:

Tensor, has the same shape and data type as *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> inv = ops.Inv()
>>> x = Tensor(np.array([0.25, 0.4, 0.31, 0.52]), mindspore.float32)
>>> output = inv(x)
>>> print(output)
[4.        2.5       3.2258065 1.923077 ]
```

mindspore.ops.Invert

class mindspore.ops.**Invert**

Flips all bits of input tensor element-wise.

Refer to *mindspore.ops.invert()* for more details.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> invert = ops.Invert()
>>> x = Tensor(np.array([25, 4, 13, 9]), mindspore.int16)
>>> output = invert(x)
>>> print(output)
[-26 -5 -14 -10]
```

mindspore.ops.Lerp

class mindspore.ops.**Lerp**

```
prim = ops.Lerp()
out = prim(input, end, weight)
```

is equivalent to

```
ops.lerp(input, end, weight)
```

Refer to *mindspore.ops.lerp()* for more details.

mindspore.ops.Log

class mindspore.ops.**Log**

```
prim = ops.Log()
out = prim(input)
```

is equivalent to

```
ops.log(input)
```

Refer to *mindspore.ops.log()* for more details.

mindspore.ops.Log1p

class mindspore.ops.**Log1p**

```
prim = ops.Log1p()
out = prim(input)
```

is equivalent to

```
ops.log1p(input)
```

Refer to *mindspore.ops.log1p()* for more details.

mindspore.ops.LogicalAnd

class mindspore.ops.**LogicalAnd**

Computes the "logical AND" of two tensors element-wise.

Refer to *mindspore.ops.logical_and()* for more details.

Inputs:

- **x** (Union[Tensor, bool]) - The first input is a bool or a tensor whose data type can be implicitly converted to bool.
- **y** (Union[Tensor, bool]) - The second input is a bool when the first input is a tensor or a tensor whose data type can be implicitly converted to bool.

Outputs:

Tensor, the shape is the same as the *x* and *y* after broadcasting, and the data type is bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([True, False, True]), mindspore.bool_)
>>> y = Tensor(np.array([True, True, False]), mindspore.bool_)
>>> logical_and = ops.LogicalAnd()
>>> output = logical_and(x, y)
>>> print(output)
[ True False False]
>>> x = Tensor(1, mindspore.bool_)
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalAnd()(x, y)
>>> print(output)
False
>>> x = True
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalAnd()(x, y)
>>> print(output)
False
>>> x = True
>>> y = Tensor(np.array([True, False]), mindspore.bool_)
>>> output = ops.LogicalAnd()(x, y)
>>> print(output)
[True False]
```

mindspore.ops.LogicalNot

class mindspore.ops.**LogicalNot**

Computes the "logical NOT" of a tensor element-wise.

Refer to *mindspore.ops.logical_not()* for more details.

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

Tensor, the shape is the same as the *x*, and the dtype is bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([True, False, True]), mindspore.bool_)
>>> logical_not = ops.LogicalNot()
>>> output = logical_not(x)
>>> print(output)
[False  True False]
```

mindspore.ops.LogicalOr

class mindspore.ops.**LogicalOr**

Computes the "logical OR" of two tensors element-wise.

Refer to *mindspore.ops.logical_or()* for more details.

Inputs:

- **x** (Union[Tensor, bool]) - The first input is a bool or a tensor whose data type can be implicitly converted to bool.
- **y** (Union[Tensor, bool]) - The second input is a bool when the first input is a tensor or a tensor whose data type can be implicitly converted to bool.

Outputs:

Tensor, the shape is the same as the *x* and *y* after broadcasting, and the data type is bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([True, False, True]), mindspore.bool_)
>>> y = Tensor(np.array([True, True, False]), mindspore.bool_)
>>> logical_or = ops.LogicalOr()
>>> output = logical_or(x, y)
>>> print(output)
[ True  True  True]
>>> x = Tensor(1, mindspore.bool_)
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalOr()(x, y)
>>> print(output)
True
>>> x = True
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalOr()(x, y)
>>> print(output)
True
>>> x = True
>>> y = Tensor(np.array([True, False]), mindspore.bool_)
>>> output = ops.LogicalOr()(x, y)
>>> print(output)
[True True]
```

mindspore.ops.LogicalXor

class mindspore.ops.**LogicalXor**

Computes the "logical XOR" of two tensors element-wise.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.logical_xor()* for more details.

Inputs:

- **x** (Union[Tensor, bool]) - The first input is a bool or a tensor whose data type can be implicitly converted to bool.
- **y** (Union[Tensor, bool]) - The second input is a bool when the first input is a tensor or a tensor whose data type can be implicitly converted to bool.

Outputs:

Tensor, the shape is the same as the *x* and *y* after broadcasting, and the data type is bool.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([True, False, True]), mindspore.bool_)
>>> y = Tensor(np.array([True, True, False]), mindspore.bool_)
>>> logical_xor = ops.LogicalXor()
>>> output = logical_xor(x, y)
>>> print(output)
[ False True True]
>>> x = Tensor(1, mindspore.bool_)
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalXor()(x, y)
>>> print(output)
True
>>> x = True
>>> y = Tensor(0, mindspore.bool_)
>>> output = ops.LogicalXor()(x, y)
>>> print(output)
True
>>> x = True
>>> y = Tensor(np.array([True, False]), mindspore.bool_)
>>> output = ops.LogicalXor()(x, y)
>>> print(output)
[False True]
```

mindspore.ops.Logit

class mindspore.ops.**Logit**(*eps=- 1.0*)

Calculate the logit of a tensor element-wise. Element in *x* is clamped to [eps, 1-eps].

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.logit()* for more details.

Parameters

eps (*float, optional*) – The epsilon. The input clamp bound is defined as [eps, 1-eps]. Default: `-1.0`.

Inputs:

- **x** (Tensor) - The input tensor of type float16, float32 or float64.

Outputs:

Tensor, with the same shape and dtype as the *x*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0.1, 0.2, 0.3]).astype(np.float32))
>>> op = ops.Logit(eps=1e-5)
>>> output = op(x)
>>> print(output)
[-2.1972246 -1.3862944 -0.8472978]
```

mindspore.ops.Mod

class mindspore.ops.**Mod**

Computes the remainder of dividing the first input tensor by the second input tensor element-wise.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, both dtypes cannot be bool, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

When the elements of input exceed 2048, the accuracy of operator cannot guarantee the requirement of double thousandths in the mini form. Due to different architectures, the calculation results of this operator on NPU and CPU may be inconsistent. If shape is expressed as $(D1, D2, ..., Dn)$, then $D1 * D2... * DN <= 1000000, n <= 8$.

Inputs:

- **x** (Union[Tensor, numbers.Number, bool]) - The first input is a number, a bool or a tensor whose data type is number.
- **y** (Union[Tensor, numbers.Number, bool]) - When the first input is a tensor, The second input could be a number, a bool or a tensor whose data type is number. When the first input is a number or a bool the second input must be a tensor whose data type is number.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** – If neither *x* nor *y* is one of the following: Tensor, number, bool.
- **TypeError** – If neither *x* nor *y* is a Tensor.
- **ValueError** – If the shape *x* and *y* cannot be broadcasted to each other.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([-4.0, 5.0, 6.0]), mindspore.float32)
>>> y = Tensor(np.array([3.0, 2.0, 3.0]), mindspore.float32)
>>> mod = ops.Mod()
>>> output = mod(x, y)
>>> print(output)
[-1.  1.  0.]
```

mindspore.ops.Mul

class mindspore.ops.**Mul**

```
prim = ops.Mul()
out = prim(input, other)
```

is equivalent to

```
ops.mul(input, other)
```

Refer to *mindspore.ops.mul()* for more details.

mindspore.ops.MulNoNan

class mindspore.ops.**MulNoNan**

Computes *x* * *y* element-wise. If *y* is zero, no matter what *x* is, it will return 0.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, the shapes of them could be broadcasted. When the inputs are one tensor and one scalar, the scalar could only be a constant.

$$output_{ij} = \begin{cases} 0, & y_{ij} = 0; \\ x_{ij} * y_{ij}, & otherwise. \end{cases}$$

Note: The shapes of *x* and *y* should be the same or can be broadcasted. This is noncommutative: if *y* is NaN or infinite and *x* is 0, the result will be NaN.

Inputs:

- **x** (Union[Tensor]) - The first input is a tensor whose data type is one of int32, int64, float16, float32, float64, complex64, complex128 currently or scalar.
- **y** (Union[Tensor]) - The second input is a tensor whose data type is one of int32, int64, float16, float32, float64, complex64, complex128 currently or scalar.

Outputs:

Tensor, the shape is the same as the shape of input Tensor after broadcasting, and the data type is the one with higher precision among the two inputs.

Raises

TypeError – If neither *x* nor *y* is a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1 : same data type and shape of two inputs, there are some 0 in y.
>>> x = Tensor(np.array([[-1.0, 6.0, np.inf], [np.nan, -7.0, 4.0]]), mindspore.float32)
>>> y = Tensor(np.array([[-1.0, 4.0, 0], [0, -3.0, 1.0]]), mindspore.float32)
>>> mul_no_nan = ops.MulNoNan()
>>> output = mul_no_nan(x, y)
>>> print(output)
[[ 1. 24. 0.]
[ 0. 21. 4.]]
>>> # case 2 : the shape of two inputs is same, there are some 0 in x, y.
>>> x = Tensor(np.array([[-1.0, 6.0, 0], [0, np.nan, 4.0]]), mindspore.float32)
>>> y = Tensor(np.array([[-1.0, 4.0, np.inf], [np.nan, 0, 1.0]]), mindspore.float32)
>>> output = mul_no_nan(x, y)
>>> print(output)
[[ 1. 24. nan]
 [nan  0. 4.]]
>>> print(output.dtype)
Float32
>>> # case 3 : the y is a scalar.
>>> x = Tensor(np.array([[-1.0, 6.0, 0], [0, np.nan, 4.0]]), mindspore.float32)
>>> y = Tensor(0, mindspore.float32)
>>> output = mul_no_nan(x, y)
>>> print(output)
[[0. 0. 0.]
 [0. 0. 0.]]
```

mindspore.ops.Neg

class mindspore.ops.**Neg**

```
prim = ops.Neg()
out = prim(input)
```

is equivalent to

```
ops.neg(input)
```

Refer to *mindspore.ops.neg()* for more details.

mindspore.ops.NextAfter

class mindspore.ops.**NextAfter**

```
prim = ops.NextAfter()
out = prim(input, other)
```

is equivalent to

```
ops.nextafter(input, other)
```

Refer to *mindspore.ops.nextafter()* for more details.

mindspore.ops.Pow

class mindspore.ops.**Pow**

```
prim = ops.Pow()
out = prim(input, exponent)
```

is equivalent to

```
ops.pow(input, exponent)
```

Refer to *mindspore.ops.pow()* for more details.

mindspore.ops.Polar

class mindspore.ops.**Polar**

Converts polar coordinates to Cartesian coordinates.

Returns a complex tensor, its elements are Cartesian coordinates constructed with the polar coordinates which is specified by radial distance *abs* and polar angle *angle*.

Refer to *mindspore.ops.polar()* for more details.

$$y_i = abs_i * \cos(angle_i) + abs_i * \sin(angle_i) * j$$

Warning: This is an experimental API that is subject to change.

Inputs:

- **abs** (Tensor, float) - Radial distance. Tensor of any dimension, with dtype required to be float32.
- **angle** (Tensor, float) - Polar angle. It has the same shape and dtype as *abs*.

Outputs:

Tensor, with the same shape as *abs* and the dtype is complex64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> x1 = Tensor(np.array([1, 2]), mindspore.float32)
>>> x2 = Tensor(np.array([3, 4]), mindspore.float32)
>>> op_polar = ops.Polar()
>>> output = op_polar(x1, x2)
>>> print(output)
[-0.9899925 +0.14112002j -1.3072872-1.5136049j]
>>> x1 = Tensor(2.1, mindspore.float32)
>>> x2 = Tensor(2.1, mindspore.float32)
>>> output = op_polar(x1, x2)
>>> print(output)
(-1.0601765+1.8127397j)
```

mindspore.ops.Polygamma

class mindspore.ops.**Polygamma**

Computes the a.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.polygamma()* for more details.

Inputs:

- **a** (Tensor) - The order of the polygamma function, it has shape (), supported types: int32, int64.
- **x** (Tensor) - The tensor to compute the a-th derivative of the polygamma function with, supported types: float16, float32, float64.

Outputs:

Tensor, has the same dtype as x.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, -0.5]), mindspore.float32)
>>> a = Tensor(np.array(1), mindspore.int64)
>>> polygamma = ops.Polygamma()
>>> output = polygamma(a, x)
>>> print(output)
[1.644934 8.934802]
>>> a = Tensor(np.array(2), mindspore.int64)
>>> output = polygamma(a, x)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[-2.404114  -0.8287967]
>>> a = Tensor(np.array(3), mindspore.int64)
>>> output = polygamma(a, x)
>>> print(output)
[  6.4939404 193.40909  ]
>>> a = Tensor(np.array(4), mindspore.int64)
>>> output = polygamma(a, x)
>>> print(output)
[-24.886265   -3.4742498]
```

mindspore.ops.Real

class mindspore.ops.**Real**

```
prim = ops.Real()
out = prim(input)
```

is equivalent to

```
ops.real(input)
```

Refer to *mindspore.ops.real()* for more details.

mindspore.ops.RealDiv

class mindspore.ops.**RealDiv**

Divides the first input tensor by the second input tensor in floating-point type element-wise.

Refer to *mindspore.ops.div()* for more details.

Inputs:

- **x** (Union[Tensor, Number, bool]) - The first input is a number or a bool or a tensor whose data type is number or bool.
- **y** (Union[Tensor, Number, bool]) - The second input is a number or a bool when the first input is a tensor or a tensor whose data type is number or bool.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([4.0, 5.0, 6.0]), mindspore.float32)
>>> realdiv = ops.RealDiv()
>>> output = realdiv(x, y)
>>> print(output)
[0.25 0.4  0.5 ]
```

mindspore.ops.Reciprocal

class mindspore.ops.**Reciprocal**

Returns reciprocal of a tensor element-wise.

$$out_i = \frac{1}{x_i}$$

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

Tensor, has the same shape as the x.

Raises

TypeError – If x is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 4.0]), mindspore.float32)
>>> reciprocal = ops.Reciprocal()
>>> output = reciprocal(x)
>>> print(output)
[1.   0.5  0.25]
```

mindspore.ops.Rint

class mindspore.ops.**Rint**

Returns an integer that is closest to *input_x* element-wise.

Inputs:

- **input_x** (Tensor) - Input tensor of any dimension, which must be one of the following types: float16, float32, float64.

Outputs:

Tensor, has the same shape and type as *input_x*.

Raises

TypeError – If dtype of *input_x* is not in [float16, float32, float64].

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([-1.6, -0.1, 1.5, 2.0]), mindspore.float32)
>>> op = ops.Rint()
>>> output = op(input_x)
>>> print(output)
[-2.  0.  2.  2.]
>>> input_x = Tensor(np.array([[-2.0, -1.9, -1.8, -1.7, -1.6],
...                            [-2.0, -1.9, -1.8, -1.7, -1.6]]), mindspore.float32)
>>> output = op(input_x)
>>> print(output)
[[-2. -2. -2. -2. -2.]
 [-2. -2. -2. -2. -2.]]
```

mindspore.ops.Round

class mindspore.ops.**Round**

Returns half to even of a tensor element-wise.

$$out_i \approx input_i$$

Note: The input data types supported by the Ascend platform include bfloat16 (Atlas training series products are not supported), float16, float32, float64, int32, and int64.

Inputs:

- **input** (Tensor) - The input tensor.
- **decimals** (int, optional) - Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point. It supports converting the single-element tensor to an int. When *input* type is int32 or int64, the *decimals* should be 0.

Outputs:

Tensor, has the same shape and type as the *input*.

Raises

- **TypeError** –If *input* is not a Tensor.
- **RuntimeError** –If *input* type is int32 or int64, the *decimals* is not 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([0.8, 1.5, 2.3, 2.5, -4.5]), mindspore.float32)
>>> round = ops.Round()
>>> output = round(input)
>>> print(output)
[ 1.  2.  2.  2. -4.]
```

mindspore.ops.Rsqrt

class mindspore.ops.**Rsqrt**

```
prim = ops.Rsqrt()
out = prim(input)
```

is equivalent to

```
ops.rsqrt(input)
```

Refer to *mindspore.ops.rsqrt()* for more details.

mindspore.ops.Sign

class mindspore.ops.**Sign**

```
prim = ops.Sign()
out = prim(input)
```

is equivalent to

```
ops.sign(input)
```

Refer to *mindspore.ops.sign()* for more details.

mindspore.ops.Sin

class mindspore.ops.**Sin**

```
prim = ops.Sin()
out = prim(input)
```

is equivalent to

```
ops.sin(input)
```

Refer to *mindspore.ops.sin()* for more details.

mindspore.ops.Sinc

class mindspore.ops.**Sinc**

```
prim = ops.Sinc()
out = prim(input)
```

is equivalent to

```
ops.sinc(input)
```

Refer to *mindspore.ops.sinc()* for more details.

mindspore.ops.Sinh

class mindspore.ops.**Sinh**

```
prim = ops.Sinh()
out = prim(input)
```

is equivalent to

```
ops.sinh(input)
```

Refer to *mindspore.ops.sinh()* for more details.

mindspore.ops.Sqrt

class mindspore.ops.**Sqrt**

```
prim = ops.Sqrt()
out = prim(x)
```

is equivalent to

```
ops.sqrt(x)
```

Refer to *mindspore.ops.sqrt()* for more details.

mindspore.ops.Square

class mindspore.ops.**Square**

```
prim = ops.Square()
out = prim(input)
```

is equivalent to

```
ops.square(input)
```

Refer to *mindspore.ops.square()* for more details.

mindspore.ops.SquaredDifference

class mindspore.ops.**SquaredDifference**

Subtracts the second input tensor from the first input tensor element-wise and returns square of it.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. The inputs must be two tensors or one tensor and one scalar. When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

$$out_i = (x_i - y_i) * (x_i - y_i) = (x_i - y_i)^2$$

Inputs:

- **x** (Union[Tensor, Number, bool]) - The first input is a number, or a bool, or a tensor.
- **y** (Union[Tensor, Number, bool]) - The second input is a number, or a bool when the first input is a tensor, or a tensor.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

TypeError – if *x* and *y* is not a Number or a bool or a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1.0, 2.0, 3.0]), mindspore.float32)
>>> y = Tensor(np.array([2.0, 4.0, 6.0]), mindspore.float32)
>>> squared_difference = ops.SquaredDifference()
>>> output = squared_difference(x, y)
>>> print(output)
[1. 4. 9.]
```

mindspore.ops.SquareSumAll

class mindspore.ops.**SquareSumAll**

Returns the square sum of a tensor element-wise.

$$\begin{cases} out_x = \sum_0^N (x_i)^2 \\ out_y = \sum_0^N (y_i)^2 \end{cases}$$

Note: SquareSumAll only supports float16 and float32 data type.

Inputs:

- **x** (Tensor) - The input tensor. The data type must be float16 or float32. $(N, *)$ where $*$ means, any number of additional dimensions.
- **y** (Tensor) - The input tensor has the same type and shape as the *x*.

Outputs:

- **output_x** (Tensor) - The same type as the *x*.
- **output_y** (Tensor) - The same type as the *x*.

Raises

- **TypeError** –If neither *x* nor *y* is a Tensor.
- **ValueError** –If *x* and *y* are not the same shape.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> x = Tensor(np.array([0, 0, 2, 0]), mindspore.float32)
>>> y = Tensor(np.array([0, 0, 2, 4]), mindspore.float32)
>>> square_sum_all = ops.SquareSumAll()
>>> output = square_sum_all(x, y)
>>> print(output)
(Tensor(shape=[], dtype=Float32, value= 4),
 Tensor(shape=[], dtype=Float32, value= 20))
```

mindspore.ops.Sub

class mindspore.ops.**Sub**

Subtracts the second input tensor from the first input tensor element-wise.

Refer to *mindspore.ops.sub()* for more details.

Note:

- When the two inputs have different shapes, they must be able to broadcast to a common shape.
- The two inputs can not be bool type at the same time, [True, Tensor(True, bool_), Tensor(np.array([True]), bool_)] are all considered bool type.
- The two inputs comply with the implicit type conversion rules to make the data types consistent.

Inputs:

- **x** (Union[Tensor, number.Number, bool]) - The first input is a number.Number or a bool or a tensor whose data type is number or bool_.
- **y** (Union[Tensor, number.Number, bool]) - The second input, when the first input is a Tensor, the second input should be a number.Number or bool value, or a Tensor whose data type is number or bool.

Outputs:

Tensor, the shape is the same as the two inputs after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1, 2, 3]), mindspore.int32)
>>> y = Tensor(np.array([4, 5, 6]), mindspore.int32)
>>> sub = ops.Sub()
>>> output = sub(x, y)
>>> print(output)
[-3 -3 -3]
```

mindspore.ops.Tan

class mindspore.ops.**Tan**

Computes tangent of *input* element-wise.

Refer to *mindspore.ops.tan()* for more details.

Inputs:

- **input** (Tensor) - Input tensor of any dimension.

Outputs:

Tensor, has the same shape as *input*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> tan = ops.Tan()
>>> x = Tensor(np.array([-1.0, 0.0, 1.0]), mindspore.float32)
>>> output = tan(x)
>>> print(output)
[-1.5574081 0. 1.5574081]
```

mindspore.ops.Trunc

class mindspore.ops.**Trunc**

```
prim = ops.Trunc()
out = prim(input)
```

is equivalent to

```
ops.trunc(input)
```

Refer to *mindspore.ops.trunc()* for more details.

mindspore.ops.TruncateDiv

class mindspore.ops.**TruncateDiv**

Divides the first input tensor by the second input tensor element-wise and rounds the results of division towards zero. Equivalent to C-style integer division.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

Note: Broadcasting is supported.

Inputs:

- **x** (Union[Tensor, Number, bool]) - The first input is a number, or a bool, or a tensor whose data type is number or bool.
- **y** (Union[Tensor, Number, bool]) - The second input is a number, or a bool when the first input is a tensor, or a tensor whose data type is number or bool.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

TypeError – If *x* and *y* is not one of the following: Tensor, Number, bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([2, 4, -1]), mindspore.int32)
>>> y = Tensor(np.array([3, 3, 3]), mindspore.int32)
>>> truncate_div = ops.TruncateDiv()
>>> output = truncate_div(x, y)
>>> print(output)
[0 1 0]
```

mindspore.ops.TruncateMod

class mindspore.ops.**TruncateMod**

Returns the remainder of division element-wise.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. When the inputs are one tensor and one scalar, the scalar could only be a constant.

Warning:

- The input data does not support 0.
- When the elements of input exceed 2048, the accuracy of operator cannot guarantee the requirement of double thousandths in the mini form.
- Due to different architectures, the calculation results of this operator on NPU and CPU may be inconsistent.
- If shape is expressed as (D1,D2⋯,Dn), then D1*D2⋯*DN<=1000000,n<=8.

Inputs:

- **x** (Union[Tensor, numbers.Number, bool]) - The first input is a number, or a bool, or a tensor whose data type is number or bool.
- **y** (Union[Tensor, numbers.Number, bool]) - The second input is a number, or a bool when the first input is a tensor, or a tensor whose data type is number or bool.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision among the two inputs.

Raises

- **TypeError** – If neither *x* nor *y* is one of the following: Tensor, number, bool.

- **TypeError** –If neither *x* nor *y* is a Tensor.
- **ValueError** –If the shape *x* and *y* cannot be broadcasted to each other.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([2, 4, -1]), mindspore.int32)
>>> y = Tensor(np.array([3, 3, 3]), mindspore.int32)
>>> truncate_mod = ops.TruncateMod()
>>> output = truncate_mod(x, y)
>>> print(output)
[ 2  1 -1]
```

mindspore.ops.Xdivy

class mindspore.ops.**Xdivy**

Divides the first input tensor by the second input tensor element-wise. Returns zero when *x* is zero.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. When the inputs are two tensors, dtypes of them cannot be bool at the same time, and the shapes of them could be broadcast. If one of the inputs is scalar, the scalar could only be a constant.

Inputs:

- **x** (Union[Tensor, Number, bool]) - The first input is a number, or a bool, or a tensor whose data type is float16, float32, float64, complex64, complex128 or bool.
- **y** (Union[Tensor, Number, bool]) - The second input is a number, or a bool when the first input is a tensor, or a tensor whose data type is float16, float32, float64, complex64, complex128 or bool.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** –If *x* and *y* is not one of the following: Tensor, Number, bool.
- **TypeError** –If dtype of *x* and 'y' is not in [float16, float32, float64, complex64, complex128, bool].
- **ValueError** –If *x* could not be broadcast to a tensor with shape of *y*.
- **RuntimeError** –If the data type of *x*, *y* conversion of Parameter is given but data type conversion of Parameter is not supported.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([2, 4, -1]), mindspore.float32)
>>> y = Tensor(np.array([2, 2, 2]), mindspore.float32)
>>> xdivy = ops.Xdivy()
>>> output = xdivy(x, y)
>>> print(output)
[ 1.   2.  -0.5]
```

mindspore.ops.Xlogy

class mindspore.ops.**Xlogy**

Computes the first input tensor multiplied by the logarithm of second input tensor element-wise. Returns zero when *input* is zero.

$$out_i = input_i \ln other_i$$

Inputs of *input* and *other* comply with the implicit type conversion rules to make the data types consistent.

Inputs:

- **input** (Tensor) - The first input is a tensor whose data type is number or bool_.
- **other** (Tensor) - The second input is a tensor whose data type is number or bool_.

Outputs:

- **y** (Tensor) - the shape is the broadcast of *input* and *other*, and the data type is the one with higher precision or higher digits among the two inputs.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *other* is not a Tensor.
- **ValueError** – If *input* and *other* can not broadcast.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.array([-5, 0, 4]), mindspore.float32)
>>> other = Tensor(np.array([2, 2, 2]), mindspore.float32)
>>> op = ops.Xlogy()
>>> output = op(input, other)
>>> print(output)
[-3.465736   0.        2.7725887]
```

mindspore.ops.Zeta

class mindspore.ops.**Zeta**

Compute the Hurwitz zeta function ζ(x,q) of input Tensor.

$$\zeta(x,q) = \sum_{n=0}^{\infty} (q+n)^{-x}$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - A Tensor, types: float32, float64.
- **q** (Tensor) - A Tensor, must have the same shape and type as *x*.

Outputs:

Tensor, has the same dtype and shape as the x.

Raises

- **TypeError** – If either of *x* and *q* is not tensor.
- **TypeError** – If dtype of *x* is neither float32 nor float64.
- **TypeError** – If dtype of *q* is neither float32 nor float64.
- **ValueError** – If shape of *x* is not same as the *q*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([10.]), mindspore.float32)
>>> q = Tensor(np.array([1.]), mindspore.float32)
>>> zeta = ops.Zeta()
>>> z = zeta(x, q)
>>> print(z)
[1.0009946]
```

18.3.9 Reduction Operator

API Name	Description	Supported Platforms
mindspore.ops.Argmax	Returns the indices of the maximum value along a specified *axis* of a Tensor.	`Ascend` `GPU` `CPU`
mindspore.ops.ArgMaxWithValue	Calculates the maximum value along with the given axis for the input tensor, and returns the maximum values and indices.	`Ascend` `GPU` `CPU`

continues on next page

Table 12 – continued from previous page

mindspore.ops.Argmin	Returns the indices of the minimum value along a specified *axis* of a Tensor.	Ascend GPU CPU
mindspore.ops.ArgMinWithValue	Calculates the minimum value along with the given axis for the input tensor, and returns the minimum values and indices.	Ascend GPU CPU
mindspore.ops.Median	Computes the median and its corresponding indices of input tensor in the *axis* dimension.	GPU CPU
mindspore.ops.ReduceAll	Refer to *mindspore.ops.all()* for more details.	Ascend GPU CPU
mindspore.ops.ReduceAny	Reduces a dimension of a tensor by the "logical OR" of all elements in the dimension, by default.	Ascend GPU CPU
mindspore.ops.ReduceMax	Reduces a dimension of a tensor by the maximum value in this dimension, by default.	Ascend GPU CPU
mindspore.ops.ReduceMean	Reduces a dimension of a tensor by averaging all elements in the dimension, by default.	Ascend GPU CPU
mindspore.ops.ReduceMin	Reduces a dimension of a tensor by the minimum value in the dimension, by default.	Ascend GPU CPU
mindspore.ops.ReduceProd	Reduces a dimension of a tensor by multiplying all elements in the dimension, by default.	Ascend GPU CPU
mindspore.ops.ReduceSum	Reduces a dimension of a tensor by summing all elements in the dimension, by default.	Ascend GPU CPU

mindspore.ops.Argmax

class mindspore.ops.**Argmax**(*axis=- 1*, *output_type=mstype.int32*)

Returns the indices of the maximum value along a specified *axis* of a Tensor.

Refer to *mindspore.ops.argmax()* for more details.

Parameters

- **axis** (*int*) – Axis where the Argmax operation applies to. Default: `-1` .
- **output_type** (*mindspore.dtype*) – Output data type. Supported types: `mstype.int32` , `mstype.int64` . Default: `mstype.int32` .

Inputs:

- **input_x** (Tensor) - The input tensor. $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, indices of the max value of input tensor across the axis.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[1, 20, 5], [67, 8, 9], [130, 24, 15]]).astype(np.float32))
>>> output = ops.Argmax(output_type=mindspore.int32)(input_x)
>>> print(output)
[1 0 0]
```

mindspore.ops.ArgMaxWithValue

class mindspore.ops.**ArgMaxWithValue**(*axis=0*, *keep_dims=False*)

Calculates the maximum value along with the given axis for the input tensor, and returns the maximum values and indices.

Note: In auto_parallel and semi_auto_parallel mode, the first output index can not be used.

Warning:

- If there are multiple maximum values, the index of the first maximum value is used.
- The value range of *axis* is [-dims, dims - 1]. "dims" is the dimension length of *input*.

Also see *mindspore.ops.max()*.

Parameters

- **axis** (*int*) – The dimension to reduce. Default: `0` .
- **keep_dims** (*bool*) – Whether to reduce dimension, if `True` , the output will keep same dimension with the input, the output will reduce dimension if `False` . Default: `False` .

Inputs:

- **input** (Tensor) - The input tensor, can be any dimension. Set the shape of input tensor as $(input_1, input_2, ..., input_N)$.

Outputs:

tuple (Tensor), tuple of 2 tensors, containing the corresponding index and the maximum value of the input tensor.

- **index** (Tensor) - The index for the maximum value of the input tensor, with dtype int64. If *keep_dims* is `True` , the shape of output tensors is $(input_1, input_2, ..., input_{axis-1}, 1, input_{axis+1}, ..., input_N)$. Otherwise, the shape is $(input_1, input_2, ..., input_{axis-1}, input_{axis+1}, ..., input_N)$.
- **values** (Tensor) - The maximum value of input tensor, with the same shape as *index*, and same dtype as *input*.

Raises

- **TypeError** – If *keep_dims* is not a bool.
- **TypeError** – If *axis* is not an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> index, output = ops.ArgMaxWithValue()(input_x)
>>> print(index, output)
3 0.7
>>> index, output = ops.ArgMaxWithValue(keep_dims=True)(input_x)
>>> print(index, output)
[3] [0.7]
```

mindspore.ops.Argmin

class mindspore.ops.**Argmin**(*axis=- 1*, *output_type=mstype.int32*)

Returns the indices of the minimum value along a specified *axis* of a Tensor.

If the shape of input tensor is $(x_1, ..., x_N)$, the shape of the output tensor is $(x_1, ..., x_{axis-1}, x_{axis+1}, ..., x_N)$.

Parameters

- **axis** (*int, optional*) – Axis where the Argmin operation applies to. Default: `-1` .
- **output_type** (*mindspore.dtype*, optional) – Output data type. Supported types: `mstype.int32` , `mstype.int64` . Default: `mstype.int32` .

Inputs:

- **input_x** (Tensor) - Input tensor. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, which is the minimum index in the specified axis of input Tensor.

Raises

- **TypeError** – If *axis* is not an int.
- **TypeError** – If *output_type* is neither int32 nor int64.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([2.0, 3.1, 1.2]), mindspore.float32)
>>> index = ops.Argmin()(input_x)
>>> print(index)
2
```

mindspore.ops.ArgMinWithValue

class mindspore.ops.**ArgMinWithValue**(*axis=0*, *keep_dims=False*)

Calculates the minimum value along with the given axis for the input tensor, and returns the minimum values and indices.

Note: In auto_parallel and semi_auto_parallel mode, the first output index can not be used.

Warning:

- If there are multiple minimum values, the index of the first minimum value is used.
- The value range of *axis* is [-dims, dims - 1]. "dims" is the dimension length of *input*.

Also see *mindspore.ops.min()*.

Parameters

- **axis** (*int*) −The dimension to reduce. Default: `0` .
- **keep_dims** (*bool*) −Whether to reduce dimension, if `True` the output will keep the same dimension as the input, the output will reduce dimension if `False` . Default: `False` .

Inputs:

- **input** (Tensor) - The input tensor, can be any dimension. Set the shape of input tensor as $(input_1, input_2, ..., input_N)$ .Complex tensor is not supported.

Outputs:

tuple (Tensor), tuple of 2 tensors, containing the corresponding index and the minimum value of the input tensor.

- **index** (Tensor) - The index for the minimum value of the input tensor, with dtype int64. If *keep_dims* is `True` , the shape of output tensors is $(input_1, input_2, ..., input_{axis-1}, 1, input_{axis+1}, ..., input_N)$. Otherwise, the shape is $(input_1, input_2, ..., input_{axis-1}, input_{axis+1}, ..., input_N)$.
- **values** (Tensor) - The minimum value of input tensor, with the same shape as *index*, and same dtype as *input*.

Raises

- **TypeError** −If *input* is not Tensor.
- **TypeError** −If *keep_dims* is not a bool.
- **TypeError** −If *axis* is not an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([0.0, 0.4, 0.6, 0.7, 0.1]), mindspore.float32)
>>> index, output = ops.ArgMinWithValue()(x)
>>> print(index, output)
0 0.0
>>> index, output = ops.ArgMinWithValue(keep_dims=True)(x)
>>> print(index, output)
[0] [0.0]
```

mindspore.ops.Median

class mindspore.ops.**Median**(*global_median=False*, *axis=0*, *keep_dims=False*, *ignore_nan=False*)

Computes the median and its corresponding indices of input tensor in the *axis* dimension. If *global_median* is True, computes the median of all elements of tensor.

Warning:

- *indices* does not necessarily contain the first occurrence of each median value found in the *input*, unless it is unique. The specific implementation of this API is device-specific. The results may be different on CPU and GPU.
- When attr *global_median* is `True` , the value of the second output tensor *indices* is meaningless.

Parameters

- **global_median** (*bool, optional*) – Whether the output tensor is the median of all input tensor elements or not. Default: `False` .
- **axis** (*int, optional*) – The specified dimension to compute median. Default: `0` .
- **keep_dims** (*bool, optional*) – Whether the output tensor need to retain *axis* dimension or not. Default: `False` .
- **ignore_nan** (*bool, optional*) – Whether to ignore the `NaN` values in input Tensor. When `False`, if the input range (determined by *global_median*) contains a `NaN` value, the corresponding element of *values* is `NaN`. When `True`, calculates the median of the remaining elements after excluding `NaN`. Default: `False` .

Inputs:

- **x** (Tensor) - A Tensor to calculate median with.

Outputs:

- **y** (Tensor) - Median, has the same dtype as the *x*.
  - If *global_median* is `True` , the *y* has only one element.
  - If *keep_dims* is `True` , the *y* has the same shape as the *x* except the size of *y* in dimension *axis* is 1.
  - Otherwise, the *y* lacks *axis* dimension than input.
- **indices** (Tensor) - Indices, Has the same shape as the *y*, with dtype int64.

Raises

- **TypeError** – If input *x* is not a Tensor.

- **TypeError** – If *global_median* , *keep_dims* or *ignore_nan* is assigned a nonboolean value.
- **TypeError** – If *axis* is not int.
- **ValueError** – If *axis* is not in range of [-x.dim, x.dim-1].

Supported Platforms:

GPU CPU

Examples

```
>>> # case 1 : common median compute
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> x = Tensor(np.array([[5, 1, 2],[3, 5, 7], [1, 6, 4]]).astype(np.int64))
>>> median = ops.Median(global_median=False, axis=0, keep_dims=False)
>>> y = median(x)
>>> print(y)
(Tensor(shape=[3], dtype=Int64, value= [3, 5, 4]), Tensor(shape=[3], dtype=Int64, value= [1,␣
↪1, 2]))
>>> # case 2 : global median compute
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> x = Tensor(np.array([[1, 7, 6],[5, 1, 3],[9, 17, 1]]).astype(np.int32))
>>> median = ops.Median(global_median=True)
>>> y = median(x)
>>> print(y)
(Tensor(shape=[], dtype=Int32, value= 5), Tensor(shape=[], dtype=Int64, value= 0))
```

mindspore.ops.ReduceAll

class mindspore.ops.**ReduceAll**(*keep_dims=False*)

```
prim = ops.ReduceAll(keep_dims)
out = prim(input, axis)
```

is equivalent to

```
ops.all(input, axis, keep_dims)
```

Refer to *mindspore.ops.all()* for more details.

mindspore.ops.ReduceAny

class mindspore.ops.**ReduceAny**(*keep_dims=False*)

Reduces a dimension of a tensor by the "logical OR" of all elements in the dimension, by default. And also can reduce a dimension of *x* along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

keep_dims (*bool, optional*) –If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Inputs:

- **x** (Tensor[bool]) - The input tensor. The dtype of the tensor to be reduced is bool.
- **axis** (Union[int, tuple(int), list(int), Tensor], optional) - The dimensions to reduce. Default: `()` , reduce all dimensions. Only constant value is allowed. Must be in the range [-rank(x), rank(x)).

Outputs:

Tensor, the dtype is bool.

- If *axis* is `()` , and *keep_dims* is `False` , the output is a 0-D tensor representing the "logical or" of all elements in the input tensor.
- If *axis* is int, set as 2, and *keep_dims* is `False` , the shape of output is $(x_1, x_3, ..., x_R)$.
- If *axis* is tuple(int), set as (2, 3), and *keep_dims* is `False` , the shape of output is $(x_1, x_4, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [2, 3], and *keep_dims* is `False` , the shape of output is $(x_1, x_4, ..., x_R)$.

Raises

- **TypeError** –If *keep_dims* is not a bool.
- **TypeError** –If *x* is not a Tensor.
- **TypeError** –If *axis* is not one of the following: int, tuple, list or Tensor.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[True, False], [True, True]]))
>>> op = ops.ReduceAny(keep_dims=True)
>>> # case 1: Reduces a dimension by the "logical OR" of all elements in the dimension.
>>> output = op(x)
>>> print(output)
[[ True]]
>>> print(output.shape)
(1, 1)
>>> # case 2: Reduces a dimension along axis 0.
>>> output = op(x, 0)
>>> print(output)
[[ True True]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = op(x, 1)
>>> print(output)
[[True]
[ True]]
>>> # case 4: input is a scalar.
>>> x = Tensor(True)
>>> op = ops.ReduceAny()
>>> output = op(x)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
True
```

mindspore.ops.ReduceMax

class mindspore.ops.**ReduceMax**(*keep_dims=False*)

Reduces a dimension of a tensor by the maximum value in this dimension, by default. And also can reduce a dimension of *x* along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

keep_dims (*bool*) – If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input tensor.
- **axis** (Union[int, tuple(int), list(int), tensor]) - The dimensions to reduce. Default: `()` , reduce all dimensions. Must be in the range [-r, r).

Outputs:

output(Tensor): has the same dtype as the *x*.

- If *axis* is `()` , and *keep_dims* is `False` , the output is a 0-D tensor representing the maximum of all elements in the input tensor.
- If *axis* is int, set as 1, and *keep_dims* is `False` , the shape of output is $(x_0, x_2, ..., x_R)$.
- If *axis* is tuple(int) or list(int), set as (1, 2), and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [1, 2], and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** – If *keep_dims* is not a bool.
- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *axis* is not one of the following: int, tuple, list or Tensor.
- **ValueError** – If *axis* is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> output = ops.ReduceMax(keep_dims=True)(x, 1)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by the maximum value of all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = ops.ReduceMax(keep_dims=True)(x, ())
>>> print(output)
[[[9.]]]
>>> print(output.shape)
(1, 1, 1)
>>> # case 2: Reduces a dimension along axis 0.
>>> output = ops.ReduceMax(keep_dims=True)(x, 0)
>>> print(output)
[[[7. 7. 7. 7. 7. 7.]
[8. 8. 8. 8. 8. 8.]
[9. 9. 9. 9. 9. 9.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = ops.ReduceMax(keep_dims=True)(x, 1)
>>> print(output)
[[[3. 3. 3. 3. 3. 3.]]
[[6. 6. 6. 6. 6. 6.]]
[[9. 9. 9. 9. 9. 9.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = ops.ReduceMax(keep_dims=True)(x, 2)
>>> print(output)
[[[1.]
[2.]
[3.]]
[[4.]
[5.]
[6.]]
[[7.]
[8.]
[9.]]]
```

mindspore.ops.ReduceMean

class mindspore.ops.**ReduceMean**(*keep_dims=False*)

Reduces a dimension of a tensor by averaging all elements in the dimension, by default. And also can reduce a dimension of *x* along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

keep_dims (*bool*) –If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input tensor.
- **axis** (Union[int, tuple(int), list(int), Tensor]) - The dimensions to reduce. Default: `()` , reduce all dimensions. Only constant value is allowed. Must be in the range [-r, r).

Outputs:

Tensor, has the same dtype as the *x*.

- If *axis* is `()` , and *keep_dims* is `False` , the output is a 0-D tensor representing the mean of all elements in the input tensor.
- If *axis* is int, set as 1, and *keep_dims* is `False` , the shape of output is $(x_0, x_2, ..., x_R)$.
- If *axis* is tuple(int) or list(int), set as (1, 2), and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [1, 2], and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *keep_dims* is not a bool.
- **TypeError** –If *x* is not a Tensor.
- **TypeError** –If *axis* is not one of the following: int, tuple, list or Tensor.
- **ValueError** –If *axis* is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> op = ops.ReduceMean(keep_dims=True)
>>> output = op(x, 1)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by averaging all elements in the dimension.
>>> x = Tensor(np.array([[[2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2], [2, 2, 2, 2, 2, 2]],
... [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
... [[6, 6, 6, 6, 6, 6], [8, 8, 8, 8, 8, 8], [10, 10, 10, 10, 10, 10]]]),
... mindspore.float32)
>>> output = op(x)
>>> print(output)
[[[5.]]]
>>> print(output.shape)
(1, 1, 1)
>>> # case 2: Reduces a dimension along the axis 0
>>> output = op(x, 0)
```

(continues on next page)

(continued from previous page)

```
>>> print(output)
[[[4. 4. 4. 4. 4. 4.]
[5. 5. 5. 5. 5. 5.]
[6. 6. 6. 6. 6. 6.]]]
>>> # case 3: Reduces a dimension along the axis 1
>>> output = op(x, 1)
>>> print(output)
[[[2. 2. 2. 2. 2. 2.]]
[[5. 5. 5. 5. 5. 5.]]
[[8. 8. 8. 8. 8. 8.]]]
>>> # case 4: Reduces a dimension along the axis 2
>>> output = op(x, 2)
>>> print(output)
[[[ 2.]
[ 2.]
[ 2.]]
[[ 4.]
[ 5.]
[ 6.]]
[[ 6.]
[ 8.]
[10.]]]
```

mindspore.ops.ReduceMin

class mindspore.ops.**ReduceMin**(*keep_dims=False*)

Reduces a dimension of a tensor by the minimum value in the dimension, by default. And also can reduce a dimension of *x* along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

keep_dims (*bool*) – If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input tensor.
- **axis** (Union[int, tuple(int), list(int), Tensor]) - The dimensions to reduce. Default: `()` , reduce all dimensions. Only constant value is allowed. Must be in the range [-r, r).

Outputs:

Tensor, has the same dtype as the *x*.

- If *axis* is `()` , and *keep_dims* is `False` , the output is a 0-D tensor representing the minimum of all elements in the input tensor.
- If *axis* is int, set as 1, and *keep_dims* is `False` , the shape of output is $(x_0, x_2, ..., x_R)$.
- If *axis* is tuple(int), set as (1, 2), and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [1, 2], and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** –If *keep_dims* is not a bool.
- **TypeError** –If *x* is not a Tensor.
- **TypeError** –If *axis* is not one of the following: int, tuple, list or Tensor.
- **ValueError** –If *axis* is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> op = ops.ReduceMin(keep_dims=True)
>>> output = op(x, 1)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by the minimum value of all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = op(x)
>>> print(output)
[[[1.]]]
>>> print(output.shape)
(1, 1, 1)
>>> # case 2: Reduces a dimension along axis 0.
>>> output = op(x, 0)
>>> print(output)
[[[1. 1. 1. 1. 1. 1.]
[2. 2. 2. 2. 2. 2.]
[3. 3. 3. 3. 3. 3.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = op(x, 1)
>>> print(output)
[[[1. 1. 1. 1. 1. 1.]]
[[4. 4. 4. 4. 4. 4.]]
[[7. 7. 7. 7. 7. 7.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = op(x, 2)
>>> print(output)
[[[1.]
[2.]
[3.]]
[[4.]
[5.]
[6.]]
[[7.]
```

(continues on next page)

(continued from previous page)

```
[8.]
[9.]]]
```

mindspore.ops.ReduceProd

class mindspore.ops.**ReduceProd**(*keep_dims=False*)

Reduces a dimension of a tensor by multiplying all elements in the dimension, by default. And also can reduce a dimension of x along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

keep_dims (*bool*) – If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input tensor.
- **axis** (Union[int, tuple(int), list(int), Tensor]) - The dimensions to reduce. Default: `()` , reduce all dimensions. Only constant value is allowed. Must be in the range [-r, r).

Outputs:

Tensor, has the same dtype as the x.

- If *axis* is `()` , and *keep_dims* is `False` , the output is a 0-D tensor representing the product of all elements in the input tensor.
- If *axis* is int, set as 1, and *keep_dims* is `False` , the shape of output is $(x_0, x_2, ..., x_R)$.
- If *axis* is tuple(int), set as (1, 2), and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [1, 2], and *keep_dims* is `False` , the shape of output is $(x_0, x_3, ..., x_R)$.

Raises

- **TypeError** – If *keep_dims* is not a bool.
- **TypeError** – If x is not a Tensor.
- **TypeError** – If *axis* is not one of the following: int, tuple, list or Tensor.
- **ValueError** – If *axis* is out of range.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> op = ops.ReduceProd(keep_dims=True)
>>> output = op(x, 1)
>>> result = output.shape
>>> print(result)
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by multiplying all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
>>> output = op(x)
>>> print(output)
[[[2.2833798e+33]]]
>>> print(output.shape)
(1, 1, 1)
>>> # case 2: Reduces a dimension along axis 0.
>>> output = op(x, 0)
>>> print(output)
[[[ 28.  28.  28.  28.  28.  28.]
[ 80.  80.  80.  80.  80.  80.]
[162. 162. 162. 162. 162. 162.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = op(x, 1)
>>> print(output)
[[[  6.   6.   6.   6.   6.   6.]]
[[120. 120. 120. 120. 120. 120.]]
[[504. 504. 504. 504. 504. 504.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = op(x, 2)
>>> print(output)
[[[1.00000e+00]
[6.40000e+01]
[7.29000e+02]]
[[4.09600e+03]
[1.56250e+04]
[4.66560e+04]]
[[1.17649e+05]
[2.62144e+05]
[5.31441e+05]]]
```

mindspore.ops.ReduceSum

class mindspore.ops.**ReduceSum**(*keep_dims=False*, *skip_mode=False*)

Reduces a dimension of a tensor by summing all elements in the dimension, by default. And also can reduce a dimension of *x* along the *axis*. Determine whether the dimensions of the output and input are the same by controlling *keep_dims*.

Note: The *axis* with tensor type is only used for compatibility with older versions and is not recommended.

Parameters

- **keep_dims** (*bool, optional*) –If `True` , keep these reduced dimensions and the length is 1. If `False` , don't keep these dimensions. Default: `False` .
- **skip_mode** (*bool, optional*) –If `True` and *axis* is empty tuple or empty list, the ReduceSum operation isn't performed, skip it. If `True` and *axis* is other values, the ReduceSum calculation is performed normally. If `False` , do reduce. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input tensor.
- **axis** (Union[int, tuple(int), list(int), Tensor]) - The dimensions to reduce. Default: `()` , reduce all dimensions when *skip_mode* is `False` . Only constant value is allowed. Must be in the range [-rank(x), rank(x)).

Outputs:

Tensor, has the same dtype as the x.

- If *axis* is `()` , *keep_dims* is `False` , and *skip_mode* is `False` , the output is a 0-D tensor representing the sum of all elements in the input tensor.
- If *axis* is `()` , and *skip_mode* is `True` , the ReduceSum operation is not performed, output tensor is equal to the input tensor.
- If *axis* is int, set as 2, and *keep_dims* is `False` , the shape of output is $(x_1, x_3, ..., x_R)$.
- If *axis* is tuple(int) or list(int), set as (2, 3), and *keep_dims* is `False` , the shape of output is $(x_1, x_4, ..., x_R)$.
- If *axis* is 1-D Tensor, set as [2, 3], and *keep_dims* is `False` , the shape of output is $(x_1, x_4, ..., x_R)$.

Raises

- **TypeError** –If *keep_dims* is not a bool.
- **TypeError** –If *skip_mode* is not a bool.
- **TypeError** –If *x* is not a Tensor.
- **ValueError** –If *axis* is None.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.randn(3, 4, 5, 6).astype(np.float32))
>>> op = ops.ReduceSum(keep_dims=True)
>>> output = op(x, 1)
>>> output.shape
(3, 1, 5, 6)
>>> # case 1: Reduces a dimension by summing all elements in the dimension.
>>> x = Tensor(np.array([[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]],
...                      [[4, 4, 4, 4, 4, 4], [5, 5, 5, 5, 5, 5], [6, 6, 6, 6, 6, 6]],
...                      [[7, 7, 7, 7, 7, 7], [8, 8, 8, 8, 8, 8], [9, 9, 9, 9, 9, 9]]]),␣
↪mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> output = op(x)
>>> print(output)
[[[270.]]]
>>> print(output.shape)
(1, 1, 1)
>>> # case 2: Reduces a dimension along axis 0.
>>> output = op(x, 0)
>>> print(output)
[[[12. 12. 12. 12. 12. 12.]
[15. 15. 15. 15. 15. 15.]
[18. 18. 18. 18. 18. 18.]]]
>>> # case 3: Reduces a dimension along axis 1.
>>> output = op(x, 1)
>>> print(output)
[[[ 6.  6.  6.  6.  6.  6.]]
[[15. 15. 15. 15. 15. 15.]]
[[24. 24. 24. 24. 24. 24.]]]
>>> # case 4: Reduces a dimension along axis 2.
>>> output = op(x, 2)
>>> print(output)
[[[ 6.]
[12.]
[18.]]
[[24.]
[30.]
[36.]]
[[42.]
[48.]
[54.]]]
```

18.3.10 Comparison Operator

API Name	Description	Supported Platforms
mindspore.ops.ApproximateEqual	Returns `True` if abs(x-y) is smaller than tolerance element-wise, otherwise False.	`Ascend` `GPU` `CPU`
mindspore.ops.Equal	Refer to *mindspore.ops.equal()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.EqualCount	Computes the number of the same elements of two tensors.	`Ascend` `GPU` `CPU`
mindspore.ops.Greater	Refer to *mindspore.ops.greater()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.GreaterEqual	Refer to *mindspore.ops.greater_equal()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.InTopK	Determines whether the targets are in the top *k* predictions.	`Ascend` `GPU` `CPU`
mindspore.ops.IsFinite	Refer to *mindspore.ops.isfinite()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.IsInf	Refer to *mindspore.ops.isinf()* for more details.	`Ascend` `CPU` `GPU`
mindspore.ops.IsNan	Determines which elements are NaN for each position.	`Ascend` `GPU` `CPU`

continues on next page

Table 13 – continued from previous page

mindspore.ops.Less	Refer to *mindspore.ops.less()* for more details.	Ascend GPU CPU
mindspore.ops.LessEqual	Refer to *mindspore.ops.less_equal()* for more details.	Ascend GPU CPU
mindspore.ops.Maximum	Refer to *mindspore.ops.maximum()* for more details.	Ascend GPU CPU
mindspore.ops.Minimum	Refer to *mindspore.ops.minimum()* for more details.	Ascend GPU CPU
mindspore.ops.NotEqual	Refer to *mindspore.ops.not_equal()* for more details.	Ascend GPU CPU
mindspore.ops.TopK	Finds values and indices of the *k* largest entries along the last dimension.	Ascend GPU CPU

mindspore.ops.ApproximateEqual

class mindspore.ops.**ApproximateEqual**(*tolerance=1e-05*)

Returns `True` if abs(x-y) is smaller than tolerance element-wise, otherwise False.

$$out_i = \begin{cases} \text{if } |x_i - y_i| < \text{tolerance}, & True \\ \text{if } |x_i - y_i| \geq \text{tolerance}, & False \end{cases}$$

where *tolerance* indicates Acceptable maximum tolerance.

Inputs of *x* and *y* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower precision data type will be converted to the relatively highest precision data type.

Parameters

tolerance (*float, optional*) – The maximum deviation that two elements can be considered equal. Default: `1e-05`.

Inputs:

- **x** (Tensor) - A tensor. Must be one of the following types: float32, float16. $(N, *)$ where $*$ means, any number of additional dimensions, its rank should be less than 8.
- **y** (Tensor) - A tensor of the same type and shape as *x*.

Outputs:

Tensor, the shape is the same as the shape of *x*, and the data type is bool.

Raises

- **TypeError** – If *tolerance* is not a float.
- **TypeError** – If the data type of *x*, *y* conversion of Parameter is given but data type conversion of Parameter is not supported.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1, 2, 3]), mindspore.float32)
>>> y = Tensor(np.array([2, 3, 6]), mindspore.float32)
>>> approximate_equal = ops.ApproximateEqual(2.)
>>> output = approximate_equal(x, y)
>>> print(output)
[ True  True  False]
```

mindspore.ops.Equal

class mindspore.ops.**Equal**

```
prim = ops.Equal()
out = prim(input, other)
```

is equivalent to

```
ops.equal(input, other)
```

Refer to *mindspore.ops.equal()* for more details.

mindspore.ops.EqualCount

class mindspore.ops.**EqualCount**

Computes the number of the same elements of two tensors.

The two input tensors must have the same data type and shape.

Inputs:

- **x** (Tensor) - The first input tensor. If the data type and shape of y are determined, then x must be the same as y, and vice versa. $(N, *)$ where $*$ means, any number of additional dimensions.
- **y** (Tensor) - The second input tensor. If the data type and shape of x are determined, then y must be the same as x, and vice versa.

Outputs:

Tensor, with the type same as input tensor and shape as $(1,)$.

Raises

- **TypeError** – If x or y is not a Tensor.
- **ValueError** – If shape of x is not equal to shape of y.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1, 2, 3]), mindspore.int32)
>>> y = Tensor(np.array([1, 2, 4]), mindspore.int32)
>>> equal_count = ops.EqualCount()
>>> output = equal_count(x, y)
>>> print(output)
[2]
```

mindspore.ops.Greater

class mindspore.ops.**Greater**

```
prim = ops.Greater()
out = prim(input, other)
```

is equivalent to

```
ops.greater(input, other)
```

Refer to *mindspore.ops.greater()* for more details.

mindspore.ops.GreaterEqual

class mindspore.ops.**GreaterEqual**

```
prim = ops.GreaterEqual()
out = prim(input, other)
```

is equivalent to

```
ops.greater_equal(input, other)
```

Refer to *mindspore.ops.greater_equal()* for more details.

mindspore.ops.InTopK

class mindspore.ops.**InTopK**(*k*)

Determines whether the targets are in the top *k* predictions.

Refer to *mindspore.ops.intopk()* for more details.

Parameters

k (*int*) −Specifies the number of top elements to be used for computing precision along the last dimension.

Inputs:

- **x1** (Tensor) - A 2D Tensor defines the predictions of a batch of samples with float16 or float32 data type.
- **x2** (Tensor) - A 1D Tensor defines the labels of a batch of samples with int32 data type. The size of *x2* must be equal to the first dimension of *x1*. The values of *x2* can not be negative and must be equal to or less than index of x1's second dimension.

Outputs:

Tensor has 1 dimension of type bool and the same shape with *x2*. For labeling sample *i* in *x2*, if the label in the first *k* predictions for sample *i* is in *x1*, then the value is `True` , otherwise `False` .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([[1, 8, 5, 2, 7], [4, 9, 1, 3, 5]]), mindspore.float32)
>>> x2 = Tensor(np.array([1, 3]), mindspore.int32)
>>> in_top_k = ops.InTopK(3)
>>> output = in_top_k(x1, x2)
>>> print(output)
[ True  False]
```

mindspore.ops.IsFinite

class mindspore.ops.**IsFinite**

```
prim = ops.IsFinite()
out = prim(input)
```

is equivalent to

```
ops.isfinite(input)
```

Refer to *mindspore.ops.isfinite()* for more details.

mindspore.ops.IsInf

class mindspore.ops.**IsInf**

```
prim = ops.IsInf()
out = prim(input)
```

is equivalent to

```
ops.isinf(input)
```

Refer to *mindspore.ops.isinf()* for more details.

mindspore.ops.IsNan

class mindspore.ops.**IsNan**

Determines which elements are NaN for each position.

Refer to *mindspore.ops.isnan()* for more details.

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

Tensor, has the same shape of input, and the dtype is bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> is_nan = ops.IsNan()
>>> x = Tensor(np.array([np.log(-1), 1, np.log(0)]), mindspore.float32)
>>> output = is_nan(x)
>>> print(output)
[ True False False]
>>> x = Tensor(2.1, mindspore.float64)
>>> output = is_nan(x)
>>> print(output)
False
```

mindspore.ops.Less

class mindspore.ops.**Less**

```
prim = ops.Less()
out = prim(input, other)
```

is equivalent to

```
ops.less(input, other)
```

Refer to *mindspore.ops.less()* for more details.

mindspore.ops.LessEqual

class mindspore.ops.**LessEqual**

```
prim = ops.LessEqual()
out = prim(input, other)
```

is equivalent to

```
ops.less_equal(input, other)
```

Refer to *mindspore.ops.less_equal()* for more details.

mindspore.ops.Maximum

class mindspore.ops.**Maximum**

```
prim = ops.Maximum()
out = prim(input, other)
```

is equivalent to

```
ops.maximum(input, other)
```

Refer to *mindspore.ops.maximum()* for more details.

mindspore.ops.Minimum

class mindspore.ops.**Minimum**

```
prim = ops.Minimum()
out = prim(input, other)
```

is equivalent to

```
ops.minimum(input, other)
```

Refer to *mindspore.ops.minimum()* for more details.

mindspore.ops.NotEqual

class mindspore.ops.**NotEqual**

```
prim = ops.NotEqual()
out = prim(input, other)
```

is equivalent to

```
ops.not_equal(input, other)
```

Refer to *mindspore.ops.not_equal()* for more details.

mindspore.ops.TopK

class mindspore.ops.**TopK**(*sorted=True*)

Finds values and indices of the *k* largest entries along the last dimension.

> **Warning:**
>
> - If sorted is set to False, it will use the aicpu operator, the performance may be reduced. In addition, due to different memory layout and traversal methods on different platforms, the display order of calculation results may be inconsistent when *sorted* is False.

If the *input_x* is a one-dimensional Tensor, finds the *k* largest entries in the Tensor, and outputs its value and index as a Tensor. values[*k*] is the *k* largest item in *input_x*, and its index is indices [*k*].

For a multi-dimensional matrix, calculates the first *k* entries in each row (corresponding vector along the last dimension), therefore:

$$values.shape = indices.shape = input.shape[:-1] + [k]$$

If the two compared elements are the same, the one with the smaller index value is returned first.

Parameters

sorted (*bool, optional*) – If `True` , the obtained elements will be sorted by the values in descending order. If `False` , the obtained elements will not be sorted. Default: `True` .

Inputs:

- **input_x** (Tensor) - Input to be computed, 0-D input is supported on GPU, but not on Ascend or CPU. supported dtypes:
  - Ascend: int8, uint8, int32, int64, float16, float32.
  - GPU: float16, float32.
  - CPU: all numeric types.
- **k** (Union(Tensor, int)) - The number of top elements to be computed along the last dimension. The supported dtype is int32 and it should be 0-D or 1-D Tensor with shape $(1,)$.

Outputs:

A tuple consisting of *values* and *indexes*.

- **values** (Tensor) - The *k* largest elements in each slice of the last dimension.
- **indices** (Tensor) - The indices of values within the last dimension of input.

Raises

- **TypeError** – If *sorted* is not a bool.
- **TypeError** – If *input_x* is not a Tensor.
- **TypeError** – If *k* is not an int.
- **TypeError** – If dtype of *input_x* is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import mindspore
>>> input_x = Tensor([1, 2, 3, 4, 5], mindspore.float16)
>>> k = 3
>>> values, indices = ops.TopK(sorted=True)(input_x, k)
>>> print((values, indices))
(Tensor(shape=[3], dtype=Float16, value= [ 5.0000e+00,  4.0000e+00,  3.0000e+00]),␣
↪Tensor(shape=[3],
  dtype=Int32, value= [4, 3, 2]))
```

18.3.11 Linear Algebraic Operator

API Name	Description	Supported Platforms
mindspore.ops.BatchMatMul	Computes matrix multiplication between two tensors by batch.	Ascend GPU CPU
mindspore.ops.BiasAdd	Returns the sum of the input Tensor and the bias Tensor.	Ascend GPU CPU
mindspore.ops.Ger	Ger product of *x1* and *x2*.	Ascend GPU CPU
mindspore.ops.MatMul	Multiplies matrix *a* and matrix *b*.	Ascend GPU CPU
mindspore.ops.MatrixInverse	Returns the inverse of the input matrix.	Ascend GPU CPU
mindspore.ops.Ormqr	Computes the matrix-matrix multiplication of a product of Householder matrices with a general matrix.	GPU
mindspore.ops.Orgqr	Calculates the explicit representation of the orthogonal matrix Q returned by *mindspore.ops.Geqrf*.	Ascend GPU CPU
mindspore.ops.Svd	Computes the singular value decompositions of one or more matrices.	Ascend GPU CPU

mindspore.ops.BatchMatMul

class mindspore.ops.**BatchMatMul**(*transpose_a=False*, *transpose_b=False*)

Computes matrix multiplication between two tensors by batch.

text{output}[···, :, :] = text{matrix}(x[···, :, :]) * text{matrix}(y[···, :, :])

The rank of the two input tensors must be at least *2*, and the two input tensors must have the same rank if the environment is GPU or CPU.

Parameters

- **transpose_a** (*bool, optional*) – If `True` , the last two dimensions of *x* is transposed before multiplication. Default: `False` .
- **transpose_b** (*bool, optional*) – If `True` , the last two dimensions of *y* is transposed before multiplication. Default: `False` .

Inputs:

- **x** (Tensor) - The first tensor to be multiplied. The shape of the tensor is $(*B, N, C)$, where $*B$ represents the batch size which can be multidimensional, N and C are the size of the last two dimensions. If *transpose_a* is `True` , its

shape must be $(*B, C, N)$.

- **y** (Tensor) - The second tensor to be multiplied. The shape of the tensor is $(*B, C, M)$. If *transpose_b* is `True` , its shape must be $(*B, M, C)$.

Outputs:

Tensor, the shape of the output tensor is $(*B, N, M)$.

Raises

- **TypeError** –If *transpose_a* or *transpose_b* is not a bool.
- **ValueError** –If length of shape of *x* is not equal to length of shape of *y* or length of shape of inputs is less than 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.ones(shape=[2, 4, 1, 3]), mindspore.float32)
>>> y = Tensor(np.ones(shape=[2, 4, 3, 4]), mindspore.float32)
>>> batmatmul = ops.BatchMatMul()
>>> output = batmatmul(x, y)
>>> print(output.shape)
(2, 4, 1, 4)
>>> x = Tensor(np.ones(shape=[2, 4, 3, 1]), mindspore.float32)
>>> y = Tensor(np.ones(shape=[2, 4, 3, 4]), mindspore.float32)
>>> batmatmul = ops.BatchMatMul(transpose_a=True)
>>> output = batmatmul(x, y)
>>> print(output.shape)
(2, 4, 1, 4)
```

mindspore.ops.BiasAdd

class mindspore.ops.**BiasAdd**(*data_format='NCHW'*)

Returns the sum of the input Tensor and the bias Tensor. Before adding, the bias Tensor will be broadcasted to be consistent with the shape of the input Tensor.

Parameters

data_format (*str, optional*) –The format of input and output data. It should be `"NHWC"` , `"NCHW"` or `"NCDHW"` . Default is `"NCHW"` .

Inputs:

- **input_x** (Tensor) - The input tensor. The shape can be 2-5 dimensions. Supported dtypes:
  - Ascend/CPU: all Number type.
  - GPU: float16, float32, int8.
- **bias** (Tensor) - The bias tensor, with shape (C). C must be the same as channel dimension C of *input_x*. It has the same type as *input_x*.

Outputs:

Tensor, with the same shape and data type as *input_x*.

Raises

- **TypeError** –If *data_format* is not a str.
- **ValueError** –If value of *data_format* is not in the range of ['NHWC','NCHW','NCDHW'].
- **TypeError** –If *input_x* or *bias* is not a Tensor.
- **TypeError** –If dtype of *input_x* or *bias* is inconsistent.
- **TypeError** –If dimension of *input_x* is not in the range [2, 5].

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.arange(6).reshape((2, 3)), mindspore.float32)
>>> bias = Tensor(np.random.random(3).reshape((3,)), mindspore.float32)
>>> bias_add = ops.BiasAdd()
>>> output = bias_add(input_x, bias)
>>> print(output.shape)
(2, 3)
```

mindspore.ops.Ger

class mindspore.ops.**Ger**

Ger product of *x1* and *x2*. Calculate the outer product of two arrays. If *x1* is a 1D Tensor of shape $(m,)$ and *x2* is a 1D Tensor of shape $(n,)$, then *output* must be a 2D Tensor of shape (m, n).

Refer to *mindspore.ops.ger()* for more details.

Inputs:

- **x1** - (Tensor) - 1-D input Tensor.
- **x2** - (Tensor) - 1-D input Tensor, has the same dtype as *x1*.

Outputs:

Tensor, output matrix with the same dtype as inputs.With *x1* shape $(m,)$ and *x2* shape of $(n,)$,the *output* has shape (m, n).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x1 = Tensor([1., 2., 3., 4.], mindspore.float32)
>>> x2 = Tensor([1., 2., 3.], mindspore.float32)
>>> ger = ops.Ger()
>>> output = ger(x1, x2)
>>> print(output)
[[ 1.  2.  3.]
 [ 2.  4.  6.]
 [ 3.  6.  9.]
 [ 4.  8. 12.]]
```

mindspore.ops.MatMul

class mindspore.ops.**MatMul**(*transpose_a=False*, *transpose_b=False*)

Multiplies matrix *a* and matrix *b*.

$$(Output)_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{ip} b_{pj}, p \in N$$

where the i, j indicates the output of the i-th row and j-th column element.

Note:

- If $N * M$ cannot be divided by 16, the performance will be poor in ascend environment.
- The dtype of inputs must be same.
- On Ascend, float64 doesn't be supported.

Parameters

- **transpose_a** (*bool, optional*) – If `True` , *a* is transposed before multiplication. Default: `False` .
- **transpose_b** (*bool, optional*) – If `True` , *b* is transposed before multiplication. Default: `False` .

Inputs:

- **a** (Tensor) - The first tensor to be multiplied. The shape of the tensor is (N, C). If *transpose_a* is `True` , its shape must be (C, N) after transpose.
- **b** (Tensor) - The second tensor to be multiplied. The shape of the tensor is (C, M). If *transpose_b* is `True` , its shape must be (M, C) after transpose.

Outputs:

Tensor, the shape of the output tensor is (N, M).

Raises

- **TypeError** – If *transpose_a* or *transpose_b* is not a bool.
- **TypeError** – If the dtype of *a* and the dtype of *b* are not the same.
- **ValueError** – If the column of matrix dimensions of *a* is not equal to the row of matrix dimensions of *b*.

- **ValueError** –If length of shape of *a* or *b* is not equal to 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> a = Tensor(np.ones(shape=[1, 3]), mindspore.float32)
>>> b = Tensor(np.ones(shape=[3, 4]), mindspore.float32)
>>> matmul = ops.MatMul()
>>> output = matmul(a, b)
>>> print(output)
[[3. 3. 3. 3.]]
```

mindspore.ops.MatrixInverse

class mindspore.ops.**MatrixInverse**(*adjoint=False*)

Returns the inverse of the input matrix. If the matrix is irreversible, an error may be reported or an unknown result may be returned.

Note: The parameter 'adjoint' is only supporting `False` right now, because complex number is not supported at present.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

adjoint (*bool*) –An optional bool. Default: `False` .

Inputs:

- **x** (Tensor) - A matrix to be calculated. The matrix must be at least two dimensions, and the last two dimensions must be the same size.

Outputs:

Tensor, has the same type and shape as input *x*.

Raises

- **TypeError** –If *adjoint* is not a bool.
- **TypeError** –If *x* is not a Tensor.
- **ValueError** –If the last two dimensions of *x* is not same size.
- **ValueError** –If the dimension of *x* is less than 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[[-0.710504  , -1.1207525],
...                       [-1.7651395 , -1.7576632]],
...                      [[ 0.52412605,  1.9070215],
...                       [ 1.3384849 ,  1.4274558]]]), mindspore.float32)
>>> matrix_inverse = ops.MatrixInverse(adjoint=False)
>>> output = matrix_inverse(x)
>>> print(output)
[[[ 2.4095478  -1.5364188 ]
  [-2.419797    0.9740167 ]]
 [[-0.79111797  1.0569006 ]
  [ 0.74180895 -0.2904787 ]]]
```

mindspore.ops.Ormqr

class mindspore.ops.**Ormqr**(*left=True*, *transpose=False*)

Computes the matrix-matrix multiplication of a product of Householder matrices with a general matrix. Multiplies a(m, n) matrix C (given by other) with a matrix Q, where Q is represented using Householder reflectors (x, tau), which is the output of *mindspore.ops.geqrf()*.

Refer to *mindspore.ops.ormqr()* for more details.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **left** (*bool, optional*) – controls the order of multiplication. If `True` , compute op(Q)*C. If `False` , compute C*op(Q). Default: `True` .
- **transpose** (*bool, optional*) – controls whether the matrix Q is conjugate transposed or not.Default: `False` .

Inputs:

- **x** (Tensor) - Tensor of shape $(*, mn, k)$ where the value of mn depending on *left*, When *left* is `True`, the value of mn is equal to m; otherwise, the value of mn is equal to n. and * is zero or more batch dimensions.
- **tau** (Tensor) - Tensor of shape $(*, min(mn, k))$ where * is zero or more batch dimensions, and its type is the same as *x*.
- **other** (Tensor) - Tensor of shape $(*, m, n)$ where * is zero or more batch dimensions, and its type is the same as *x*.

Outputs:

- **y** (Tensor) - the output Tensor, has the same shape and data type as *other*.

Raises

- **TypeError** – If *x* or *tau* or *other* is not Tensor.
- **TypeError** – If dtype of *x* or *tau* or *other* is not one of: float64, float32, complex64, complex128.
- **ValueError** – If *x* or *other* is less than 2D.

- **ValueError** –If rank(x) - rank(tau) != 1.
- **ValueError** –If tau.shape[:-1] != x.shape[:-2]
- **ValueError** –If other.shape[:-2] != x.shape[:-2]
- **ValueError** –If left == True, other.shape[-2] < tau.shape[-1].
- **ValueError** –If left == True, other.shape[-2] != x.shape[-2].
- **ValueError** –If left == False, other.shape[-1] < tau.shape[-1].
- **ValueError** –If left == False, other.shape[-1] != x.shape[-2].

Supported Platforms:

GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-114.6, 10.9, 1.1], [-0.304, 38.07, 69.38], [-0.45, -0.17, 62]]),␣
↪mindspore.float32)
>>> tau = Tensor(np.array([1.55, 1.94, 3.0]), mindspore.float32)
>>> other = Tensor(np.array([[-114.6, 10.9, 1.1],
...                          [-0.304, 38.07, 69.38],
...                          [-0.45, -0.17, 62]]), mindspore.float32)
>>> net = ops.Ormqr()
>>> y = net(x, tau, other)
>>> print(y)
[[  63.82713   -13.823125 -116.28614 ]
 [ -53.659264  -28.157839  -70.42702 ]
 [ -79.54292    24.00183   -41.34253 ]]
```

mindspore.ops.Orgqr

class mindspore.ops.**Orgqr**

Calculates the explicit representation of the orthogonal matrix Q returned by *mindspore.ops.Geqrf*.

Warning: This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.orgqr()* for more details.

Inputs:

- **x** (Tensor) - Tensor of shape $(*, M, N)$, indicating 2D or 3D matrices, with float32, float64, complex64 and complex128 data type.
- **tau** (Tensor) - Indicates the reflecting coefficient in Householder transformation, it has shape $(*, K)$, where K is less than or equal to N, and it has the same type as x.

Outputs:

Tensor, has the same shape and data type as x.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[-114.6, 10.9, 1.1], [-0.304, 38.07, 69.38], [-0.45, -0.17, 62.]]),␣
↪mindspore.float32)
>>> tau = Tensor(np.array([1.55, 1.94, 0.0]), mindspore.float32)
>>> net = ops.Orgqr()
>>> y = net(x, tau)
>>> print(y)
[[-0.54999995 -0.2128925   0.8137956 ]
 [ 0.47119996 -0.8752807   0.08240613]
 [ 0.69749993  0.42560163  0.57772595]]
```

mindspore.ops.Svd

class mindspore.ops.**Svd**(*full_matrices=False*, *compute_uv=True*)

Computes the singular value decompositions of one or more matrices.

Refer to *mindspore.ops.svd()* for more details.

Parameters

- **full_matrices** (*bool, optional*) – If `True` , compute full-sized U and V. If `False`, compute only the leading P singular vectors, with P is the minimum of M and N. Default: `False` .
- **compute_uv** (*bool, optional*) – If `True` , compute the left and right singular vectors. If `False` , compute only the singular values. Default: `True` .

Inputs:

- **input** (Tensor) - Tensor of the matrices to be decomposed. The shape should be $(*, M, N)$, the supported dtype are float32 and float64.

Outputs:

- **s** (Tensor) - Singular values. The shape is $(*, P)$.
- **u** (Tensor) - Left singular vectors. If *compute_uv* is `False` , u will be zero value. The shape is $(*, M, P)$. If *full_matrices* is `True` , the shape will be $(*, M, M)$.
- **v** (Tensor) - Right singular vectors. If *compute_uv* is `False` , v will be zero value. The shape is $(*, N, P)$. If *full_matrices* is `True` , the shape will be $(*, N, N)$.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> ms.set_device(device_target="CPU")
>>> svd = ops.Svd(full_matrices=True, compute_uv=True)
>>> a = Tensor(np.array([[1, 2], [-4, -5], [2, 1]]).astype(np.float32))
>>> s, u, v = svd(a)
>>> print(s)
[7.0652843 1.040081 ]
>>> print(u)
[[ 0.30821905 -0.48819482 0.81649697]
 [-0.90613353  0.11070572 0.40824813]
 [ 0.2896955   0.8656849  0.4082479 ]]
>>> print(v)
[[ 0.63863593 0.769509  ]
 [ 0.769509  -0.63863593]]
```

18.4 Tensor Operation Operator

18.4.1 Tensor Construction

API Name	Description	Supported Platforms
mindspore.ops.Eps	Create a Tensor with the same data type and shape as input, and the element value is the minimum value that the corresponding data type can express.	Ascend GPU CPU
mindspore.ops.Eye	Returns a tensor with ones on the diagonal and zeros in the rest.	Ascend GPU CPU
mindspore.ops.Fill	The Fill interface is deprecated, please use the *mindspore.ops.FillV2* instead.	Deprecated
mindspore.ops.LinSpace	Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].	Ascend GPU CPU
mindspore.ops.OneHot	Computes a one-hot tensor.	Ascend GPU CPU
mindspore.ops.Ones	Creates a tensor filled with value ones.	Ascend GPU CPU
mindspore.ops.OnesLike	Returns a Tensor with a value of 1 and its shape and data type is the same as the input.	Ascend GPU CPU
mindspore.ops.Zeros	Zeros will be deprecated in the future.	Ascend GPU CPU
mindspore.ops.ZerosLike	Returns a Tensor with a value of 0 and its shape and data type is the same as the input.	Ascend GPU CPU

mindspore.ops.Eps

class mindspore.ops.**Eps**

Create a Tensor with the same data type and shape as input, and the element value is the minimum value that the corresponding data type can express.

Refer to *mindspore.ops.eps()* for more detail.

Inputs:

- **x** (Tensor) - Tensor of any dimension used to obtain the minimum value that its data type can express. The data type must be float16, float32 or float64.

Outputs:

Tensor, has the same type and shape as *x*, but filled with *x* dtype minimum val.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If data type of *x* is neither float16, float32, nor float64.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([4, 1, 2, 3], mindspore.float32)
>>> output = ops.Eps()(x)
>>> print(output)
[1.1920929e-07 1.1920929e-07 1.1920929e-07 1.1920929e-07]
```

mindspore.ops.Eye

class mindspore.ops.**Eye**

Returns a tensor with ones on the diagonal and zeros in the rest.

Inputs:

- **n** (int) - The number of rows returned.
- **m** (int) - The number of columns returned.
- **t** (mindspore.dtype) - The data type returned.

Outputs:

Tensor

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> eye = mindspore.ops.Eye()
>>> output = eye(3, 3, mindspore.int32)
>>> print(output)
[[1 0 0]
 [0 1 0]
 [0 0 1]]
>>>
>>> output = eye(3, 4, mindspore.float32)
>>> print(output)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
```

mindspore.ops.Fill

class mindspore.ops.**Fill**

The Fill interface is deprecated, please use the *mindspore.ops.FillV2* instead.

Supported Platforms:

Deprecated

mindspore.ops.LinSpace

class mindspore.ops.**LinSpace**

Generate a one-dimensional tensor with *steps* elements, evenly distributed in the interval [start, end].

$$step = (end - start)/(steps - 1)$$
$$output = [start, start + step, start + 2 * step, ..., end]$$

Inputs:

- **start** (Tensor) - Start value of interval.
- **stop** (Tensor) - Last value of interval.
- **num** (Union[int, Tensor]) - Number of elements.

Outputs:

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> start = mindspore.tensor(3, mindspore.float32)
>>> stop = mindspore.tensor(10, mindspore.float32)
>>> num = 5
>>> output = mindspore.ops.LinSpace()(start, stop, num)
>>> print(output)
[ 3.    4.75  6.5   8.25 10.  ]
>>>
>>> start = mindspore.tensor(-10, mindspore.float32)
>>> stop = mindspore.tensor(10, mindspore.float32)
>>> num = 5
>>> output = mindspore.ops.LinSpace()(start, stop, num)
>>> print(output)
[-10.  -5.   0.   5.  10.]
>>>
>>> start = mindspore.tensor(-10, mindspore.float32)
>>> stop = mindspore.tensor(10, mindspore.float32)
>>> num = 1
>>> output = mindspore.ops.LinSpace()(start, stop, num)
>>> print(output)
[-10.]
```

mindspore.ops.OneHot

class mindspore.ops.**OneHot**(*axis=-1*)

Computes a one-hot tensor.

The locations represented by indices in *indices* take value *on_value*, while all other locations take value *off_value*.

Note: If the input indices is rank *N*, the output will have rank *N+1*. The new axis is created at dimension *axis*. On Ascend, if *on_value* is Int64 dtype, *indices* must be Int64 dtype, and the value for *on_value* and *off_value* can only be 1 and 0.

Parameters

axis (*int*) – Position to insert the value. e.g. If shape of *indices* is (N, C), and *axis* is -1, the output shape will be (N, C, D), If *axis* is 0, the output shape will be (D, N, C). Default: `-1` .

Inputs:

- **indices** (Tensor) - A tensor of indices. Tensor of shape $(X_0, \ldots, X_n)$. Data type must be int32 or int64.
- **depth** (Union[int, Tensor]) - A scalar defining the depth of the one-hot dimension.
- **on_value** (Tensor) - A value to fill in output when *indices[j] = i*.
- **off_value** (Tensor) - A value to fill in output when *indices[j] != i*. It has the same data type as *on_value*.

Outputs:

Tensor, one-hot tensor. Tensor of shape $(X_0, \ldots, X_{axis}, \text{depth}, X_{axis+1}, \ldots, X_n)$.

Raises

- **TypeError** – If *axis* or *depth* is not an int.
- **TypeError** – If dtype of *on_value* is not int32, int64, float16 or float32.

- **TypeError** – If dtype of *indices* is not int32 or int64.
- **TypeError** – If *indices*, *on_value* or *off_value* is not a Tensor.
- **ValueError** – If *axis* is not in range [-1, len(indices_shape)].
- **ValueError** – If *depth* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> indices = Tensor(np.array([0, 1, 2]), mindspore.int32)
>>> depth, on_value, off_value = 3, Tensor(1.0, mindspore.float32), Tensor(0.0, mindspore.
↪float32)
>>> onehot = ops.OneHot()
>>> output = onehot(indices, depth, on_value, off_value)
>>> print(output)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

mindspore.ops.Ones

class mindspore.ops.**Ones**

Creates a tensor filled with value ones.

Refer to *mindspore.ops.ones()* for more details.

> **Warning:** For argument *size*, Tensor type input will be deprecated in the future version.

Inputs:

- **shape** (Union[tuple[int], List[int], int, Tensor]) - The specified shape of output tensor.
- **type** (*mindspore.dtype*) - The specified type of output tensor.

Outputs:

Tensor, whose dtype and size are defined by input.

Raises

TypeError – If *shape* is neither an int nor a tuple/list/Tensor of int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> ones = ops.Ones()
>>> output = ones((2, 2), mindspore.float32)
>>> print(output)
[[1. 1.]
 [1. 1.]]
>>> output = ones((3, 3), mindspore.float32)
>>> print(output)
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

mindspore.ops.OnesLike

class mindspore.ops.**OnesLike**

Returns a Tensor with a value of 1 and its shape and data type is the same as the input.

Refer to *mindspore.ops.ones_like()* for more details.

Inputs:

- **input_x** (Tensor) - Tensor of any dimension.

Outputs:

Tensor, has the same shape and type as *input_x* but filled with ones.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.int32))
>>> output = ops.OnesLike()(input_x)
>>> print(output)
[[1 1]
[1 1]]
```

mindspore.ops.Zeros

class mindspore.ops.**Zeros**

Zeros will be deprecated in the future. Please use class *mindspore.ops.zeros()* instead.

Creates a tensor filled with value zeros.

Creates a tensor with shape described by the first argument and fills it with value zeros in type of the second argument.

Warning: For argument *size*, Tensor type input will be deprecated in the future version.

Inputs:

- **shape** (tuple[int], List[int], int, Tensor) - The specified shape of output tensor.
- **type** (mindspore.dtype) - The specified type of output tensor.

Outputs:

Tensor, whose dtype and size are defined by input.

Raises

TypeError – If *shape* is neither an int nor a tuple/list/Tensor of int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> zeros = ops.Zeros()
>>> output = zeros((2, 2), mindspore.float32)
>>> print(output)
[[0. 0.]
 [0. 0.]]
```

mindspore.ops.ZerosLike

class mindspore.ops.**ZerosLike**

Returns a Tensor with a value of 0 and its shape and data type is the same as the input.

Inputs:

- **input_x** (Tensor) - Input Tensor of any dimension.

Outputs:

Tensor, has the same shape and data type as *input_x* but filled with zeros.

Raises

TypeError – If *input_x* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> zeroslike = ops.ZerosLike()
>>> input_x = Tensor(np.array([[0, 1], [2, 1]]).astype(np.float32))
>>> output = zeroslike(input_x)
>>> print(output)
[[0. 0.]
 [0. 0.]]
```

18.4.2 Random Generation Operator

API Name	Description	Supported Platforms
mindspore.ops.Bernoulli	Randomly set the elements of output to 0 or 1 with the probability of P which follows the Bernoulli distribution.	GPU CPU
mindspore.ops.Gamma	Produces random positive floating-point values x, distributed according to probability density function:	Ascend
mindspore.ops.Multinomial	Returns a tensor sampled from the multinomial probability distribution located in the corresponding row of tensor input.	Ascend GPU CPU
mindspore.ops.MultinomialWithReplacement	Returns a tensor where each row contains *numsamples* indices sampled from the multinomial distribution with replacement.	CPU
mindspore.ops.RandomCategorical	Generates random samples from a given categorical distribution tensor.	Ascend GPU CPU
mindspore.ops.RandomChoiceWithMask	Generates a random sample as index tensor with a mask tensor from a given tensor.	Ascend GPU CPU
mindspore.ops.RandomGamma	Produces random positive floating-point values x, distributed according to probability density function:	CPU
mindspore.ops.RandomPoisson	Produces random non-negative values i, distributed according to discrete probability function:	GPU CPU
mindspore.ops.Randperm	Generates n random samples from 0 to n-1 without repeating.	Ascend GPU
mindspore.ops.RandpermV2	Refer to *mindspore.ops.randperm()* for more details.	CPU
mindspore.ops.StandardLaplace	Generates random numbers according to the Laplace random number distribution (mean=0, lambda=1).	Ascend GPU CPU
mindspore.ops.StandardNormal	Generates random numbers according to the standard Normal (or Gaussian) random number distribution.	Ascend GPU CPU
mindspore.ops.UniformInt	Produces random integer values i, uniformly distributed on the closed interval [minval, maxval), that is, distributed according to the discrete probability function:	Ascend GPU CPU
mindspore.ops.UniformReal	Produces random floating-point values, uniformly distributed to the interval [0, 1).	Ascend GPU CPU

mindspore.ops.Bernoulli

class mindspore.ops.**Bernoulli**(*seed=-1*, *offset=0*)

Randomly set the elements of output to 0 or 1 with the probability of P which follows the Bernoulli distribution.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.bernoulli()* for more details.

Parameters

- **seed** (*int, optional*) – The seed value for random generating. The value of *seed* must be -1 or a positive integer, and -1 means using the current timestamp. Default: `-1` .
- **offset** (*int, optional*) – Used to change the starting position during the generation of random number sequence. Default: `0` .

Inputs:

- **x** (Tensor) - Input Tensor.
- **p** (Union[Tensor, float], optional) - Success probability, representing the probability of setting 1 for the corresponding position of the current Tensor. It has the same shape as *x*, the value of *p* must be in the range *[0, 1]*. Default: `0.5` .

Outputs:

- **y** (Tensor) - with the same shape and type as *x* .

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input_x = Tensor([0.1, 0.2, 0.3], mindspore.float32)
>>> bernoulli = ops.Bernoulli()
>>> output = bernoulli(input_x, Tensor([1.0]))
>>> print(output)
[1. 1. 1.]
>>> input_p = Tensor([0.0, 1.0, 1.0], mindspore.float32)
>>> output = bernoulli(input_x, input_p)
>>> print(output)
[0. 1. 1.]
```

mindspore.ops.Gamma

class mindspore.ops.**Gamma**(*seed=0*, *seed2=0*)

Produces random positive floating-point values x, distributed according to probability density function:

$$\mathrm{P}(x|,) = \frac{\exp(-x/)}{\cdot\,\Gamma()} \cdot x^{-1}$$

Note:

- Random Seed: Through some complex mathematical algorithms, a set of regular random numbers can be obtained, and the random seed is the initial value of this random number. The random seed is the same, the random number obtained will not change.
- When neither the global random seed nor the operator layer random seed is set or both are 0: generate completely random numbers.
- When the global random seed is set, but the operator layer random seed is not set: splice the global random seed with 0.
- When the global random seed is not set, but the operator layer random seed is set: splice 0 with the operator layer random seed.
- When both the global random seed and the operator layer random seed are set: splice the global random seed with the operator layer random seed.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) – The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) – The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (tuple) - The shape of random tensor to be generated. Only constant value is allowed.
- **alpha** (Tensor) - α is the shape parameter of Gamma distribution, which mainly determines the shape of the curve. It must be greater than 0. The data type is float32.
- **beta** (Tensor) - β is the inverse scale parameter of the Gamma distribution, which mainly determines how steep the curve is. It must be greater than 0. The data type is float32.

Outputs:

Tensor. The shape must be the broadcasted shape of Input "shape" and shapes of *alpha* and *beta*. The dtype is float32.

Raises

- **TypeError** – If data type of *seed* or *seed2* is not int.
- **TypeError** – If *alpha* or *beta* is not a Tensor.
- **TypeError** – If data type of *alpha* or *beta* is not float32.
- **ValueError** – If *shape* is not a constant value.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> shape = (3, 1, 2)
>>> alpha = Tensor(np.array([[3, 4], [5, 6]]), mstype.float32)
>>> beta = Tensor(np.array([1.0]), mstype.float32)
>>> gamma = ops.Gamma(seed=3)
>>> output = gamma(shape, alpha, beta)
>>> result = output.shape
>>> print(result)
(3, 2, 2)
```

mindspore.ops.Multinomial

class mindspore.ops.**Multinomial**(*seed=0*, *seed2=0*, *dtype=mstype.int32*)

Returns a tensor sampled from the multinomial probability distribution located in the corresponding row of tensor input.

Note:

- The rows of input do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum.
- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) –The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) –The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .
- **dtype** (mindspore.dtype, *optional*) –The type of output, must be `mstype.int32` or `mstype.int64`. Default: `mstype.int32`.

Inputs:

- **x** (Tensor) - the input tensor containing the cumsum of probabilities, must be 1 or 2 dimensions.
- **num_samples** (int) - number of samples to draw, must be a nonnegative number.

Outputs:

Tensor with the same rows as *x*, each row has num_samples sampled indices.

Raises

- **TypeError** –If neither *seed* nor *seed2* is an int.
- **TypeError** –If dtype of *num_samples* is not int.
- **TypeError** –If *dtype* is not mstype.int32 or mstype.int64.
- **ValueError** –If *seed* or *seed2* is less than 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor([[0., 9., 4., 0.]], mstype.float32)
>>> multinomial = ops.Multinomial(seed=10)
>>> output = multinomial(x, 2)
>>> print(output)
[[1 1]]
```

mindspore.ops.MultinomialWithReplacement

class mindspore.ops.**MultinomialWithReplacement**(*numsamples*, *replacement=False*)

Returns a tensor where each row contains *numsamples* indices sampled from the multinomial distribution with replacement. It diffs from *Multinomial* in that it allows the same outcome to be chosen multiple times.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.multinomial_with_replacement()* for more details.

Note: The rows of input do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum.

Parameters

- **numsamples** (*int*) –number of samples to draw, must be a nonnegative number.
- **replacement** (*bool, optional*) –Whether to draw with replacement or not. Default: `False` .

Inputs:

- **x** (Tensor) - the input tensor containing the cumsum of probabilities, must be 1 or 2 dimensions.
- **seed** (Tensor) - If *seed* and 'offset' are both set to 0, the random number generator is generated by a random seed. Otherwise, it is seeded by the given seed and offset. Supported dtype: int64.
- **offset** (Tensor) - Offset used to avoid seed collision. Supported dtype: int64.

Outputs:

Tensor with the same rows as *x*, each row has *numsamples* sampled indices.

Supported Platforms:

`CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor([[0., 9., 4., 0.]], mstype.float32)
>>> seed = Tensor(2, mstype.int64)
>>> offset = Tensor(5, mstype.int64)
>>> multinomialwithreplacement = ops.MultinomialWithReplacement(numsamples=2,
 ↪replacement=True)
>>> output = multinomialwithreplacement(x, seed, offset)
>>> print(output)
[[1 1]]
```

mindspore.ops.RandomCategorical

class mindspore.ops.**RandomCategorical**(*dtype=mstype.int64*)

Generates random samples from a given categorical distribution tensor.

> **Warning:** The Ascend backend does not support the reproducibility of random numbers, so the *seed* parameter has no effect.

Parameters

dtype (*mindspore.dtype, optional*) – The type of output. Its value must be one of mstype.int16, mstype.int32 and mstype.int64. Default: `mstype.int64`.

Inputs:

- **logits** (Tensor) - The input tensor. 2-D Tensor with shape $(batch\_size, num\_classes)$.
- **num_sample** (int) - Number of sample to be drawn. Only constant values is allowed.
- **seed** (int) - Random seed. Default: `0`. Only constant values is allowed.

Outputs:

- **output** (Tensor) - The output Tensor with shape $(batch\_size, num\_samples)$.

Raises

- **TypeError** – If *dtype* is not one of the following: mstype.int16, mstype.int32, mstype.int64.
- **TypeError** – If *logits* is not a Tensor.
- **TypeError** – If neither *num_sample* nor *seed* is an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import nn, ops, Tensor
>>> class Net(nn.Cell):
...   def __init__(self, num_sample):
...     super(Net, self).__init__()
...     self.random_categorical = ops.RandomCategorical(mindspore.int64)
...     self.num_sample = num_sample
...   def construct(self, logits, seed=0):
...     return self.random_categorical(logits, self.num_sample, seed)
...
>>> x = np.random.random((10, 5)).astype(np.float32)
>>> net = Net(8)
>>> output = net(Tensor(x))
>>> result = output.shape
>>> print(result)
(10, 8)
```

mindspore.ops.RandomChoiceWithMask

class mindspore.ops.**RandomChoiceWithMask**(*count=256*, *seed=0*, *seed2=0*)

Generates a random sample as index tensor with a mask tensor from a given tensor.

Refer to *mindspore.ops.choice_with_mask()* for more details.

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Parameters

- **count** (*int, optional*) –Number of items expected to get and the number must be greater than 0. Default: `256`.
- **seed** (*int, optional*) –The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0`.
- **seed2** (*int, optional*) –The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0`.

Inputs:

- **input_x** (Tensor[bool]) - The input tensor. The input tensor rank must be greater than or equal to 1 and less than or equal to 5.

Outputs:

Two tensors, the first one is the index tensor and the other one is the mask tensor.

- **index** (Tensor) - The output shape is 2-D, its shape is $(count, rank of input_x)$.

- **mask** (Tensor) - The output shape is 1-D, its shape is (*count*).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> rnd_choice_mask = ops.RandomChoiceWithMask()
>>> input_x = Tensor(np.ones(shape=[240000, 4]).astype(np.bool_))
>>> output_y, output_mask = rnd_choice_mask(input_x)
>>> result = output_y.shape
>>> print(result)
(256, 2)
>>> result = output_mask.shape
>>> print(result)
(256,)
```

mindspore.ops.RandomGamma

class mindspore.ops.**RandomGamma**(*seed=0*, *seed2=0*)

Produces random positive floating-point values x, distributed according to probability density function:

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Parameters

- **seed** (*int, optional*) – The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) – The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (Tensor) - The shape of random tensor to be generated. It must be constant value.
- **alpha** (Tensor) - α is the shape parameter of RandomGamma distribution, it mainly determines the shape of the graph curve. It must be greater than 0 and have date type float32.

Outputs:

Tensor. The shape should be equal to the concat shape between the input *shape* and *alpha*. The dtype is the same type as alpha.

Raises

- **TypeError** – If data type of *seed* or *seed2* is not int.

- **TypeError** –If *shape* or *alpha* is not a Tensor.
- **TypeError** –If data type of *alpha* is not float32.
- **ValueError** –If *shape* is not a constant value.

Supported Platforms:

`CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> shape = Tensor(np.array([3, 1, 2]), mstype.int32)
>>> alpha = Tensor(np.array([[3, 4], [5, 6]]), mstype.float32)
>>> gamma = ops.RandomGamma(seed=3)
>>> output = gamma(shape, alpha)
>>> result = output.shape
>>> print(result)
(3, 1, 2, 2, 2)
```

mindspore.ops.RandomPoisson

class mindspore.ops.**RandomPoisson**(*seed=0*, *seed2=0*, *dtype=mstype.int64*)

Produces random non-negative values i, distributed according to discrete probability function:

$$\mathrm{P}(i|) = \frac{\exp(-)^i}{i!}$$

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Parameters

- **seed** (*int, optional*) –The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) –The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .
- **dtype** (`mindspore.dtype`, *optional*) –The type of output. Default: `mstype.int64` .

Inputs:

- **shape** (Tensor) - The shape of random tensor to be generated, 1-D Tensor, whose dtype must be in [int32, int64].
- **rate** (Tensor) - μ parameter the distribution was constructed with. The parameter defines mean number of occurrences of the event. Its type must be in [float16, float32, float64, int32, int64].

Outputs:

Tensor. Its shape is $(*shape, *rate.shape)$. Its type is specified by *dtype*.

Raises

- **TypeError** – If *shape* is not a Tensor or its dtype is not int32 or int64.
- **TypeError** – If *dtype* is not int32 or int64.
- **ValueError** – If *shape* is not a 1-D tensor.
- **ValueError** – If *shape* elements are negative.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> shape = Tensor(np.array([2, 3]), mstype.int32)
>>> rate = Tensor(np.array([2, 2]), mstype.int32)
>>> seed = 0
>>> seed2 = 0
>>> random_poisson = ops.RandomPoisson(seed=seed, seed2=seed2)
>>> output = random_poisson(shape,rate)
>>> print(output.shape)
(2, 3, 2)
```

mindspore.ops.Randperm

class mindspore.ops.**Randperm**(*max_length=1*, *pad=- 1*, *dtype=mstype.int32*)

Generates n random samples from 0 to n-1 without repeating. If *max_length* > n, the last *max_length-n* elements will be filled with *pad*.

Parameters

- **max_length** (*int*) – Number of items expected to get and the number must be greater than 0. Default: `1` .
- **pad** (*int*) – The pad value to be filled. Default: `-1` .
- **dtype** (mindspore.dtype) – The type of output. Default: `mstype.int32` .

Inputs:

- **n** (Tensor) - The input tensor with shape $(1,)$ with and dtype int32 or int64. *n* must be in range [0, *max_length*].

Outputs:

- **output** (Tensor) - The output Tensor with shape: (*max_length*,) and type: *dtype*.

Raises

- **TypeError** – If neither *max_length* nor *pad* is an int.
- **TypeError** – If *n* is not a Tensor.

- **TypeError** – If *n* has non-Int elements.
- **TypeError** – If *n* has negative elements.
- **TypeError** – If *dtype* is not supported.
- **ValueError** – If *n* is out of range of *dtype*.
- **ValueError** – If *n* is larger than *max_length*.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> # The result of every execution is different because this operator will generate n␣
↪random samples.
>>> randperm = ops.Randperm(max_length=30, pad=-1)
>>> n = Tensor([20], dtype=mindspore.int32)
>>> output = randperm(n)
>>> print(output)
[15 6 11 19 14 16 9 5 13 18 4 10 8 0 17 2 1 12 3 7
 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

mindspore.ops.RandpermV2

class mindspore.ops.**RandpermV2**(*seed=0*, *offset=0*, *dtype=mstype.int64*)

```
prim = ops.RandpermV2(seed, offset, dtype)
out = prim(n)
```

is equivalent to

```
ops.randperm(n, seed, offset, dtype)
```

Refer to *mindspore.ops.randperm()* for more details.

mindspore.ops.StandardLaplace

class mindspore.ops.**StandardLaplace**(*seed=0*, *seed2=0*)

Generates random numbers according to the Laplace random number distribution (mean=0, lambda=1). It is defined as:

$$\mathrm{f}(x) = \frac{1}{2}\exp(-|x|)$$

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) –The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) –The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (Union[tuple[int], Tensor[int]]) - The shape of random tensor to be generated. Only constant value is allowed when the input type is tuple. And the operator supports dynamic shape only when the input type is Tensor.

Outputs:

Tensor. The shape that the input 'shape' denotes. The dtype is float32.

Raises

- **TypeError** –If seed or seed2 is not an int.
- **TypeError** –If shape is neither a tuple nor a Tensor.
- **ValueError** –If seed or seed2 is not a non-negative int.
- **ValueError** –If shape is a tuple containing non-positive items.
- **ValueError** –If shape is a Tensor, and the rank of the Tensor is not equal to 1.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> from mindspore import ops
>>> shape = (4, 16)
>>> stdlaplace = ops.StandardLaplace(seed=2)
>>> output = stdlaplace(shape)
>>> result = output.shape
>>> print(result)
(4, 16)
```

mindspore.ops.StandardNormal

class mindspore.ops.**StandardNormal**(*seed=0*, *seed2=0*)

Generates random numbers according to the standard Normal (or Gaussian) random number distribution.

Refer to *mindspore.ops.standard_normal()* for more details.

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) –The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) –The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (tuple) - The shape of random tensor to be generated. Only constant value is allowed. Supported dtypes: int32, int64.

Outputs:

Tensor. The shape is the same as the input *shape*. The dtype is float32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> shape = (3, 4)
>>> stdnormal = ops.StandardNormal(seed=2)
>>> output = stdnormal(shape)
>>> print(output)
[[-1.3031056   0.64198005 -0.65207404 -1.767485  ]
 [-0.91792876  0.6508565  -0.9098478  -0.14092612]
 [ 0.7806437   1.1585592   1.9676613  -0.00440959]]
```

mindspore.ops.UniformInt

class mindspore.ops.**UniformInt**(*seed=0*, *seed2=0*)

Produces random integer values i, uniformly distributed on the closed interval [minval, maxval), that is, distributed according to the discrete probability function:

$$\mathrm{P}(i|a,b) = \frac{1}{b-a+1},$$

where the a indicates the min distribution parameter, the b indicates the max distribution parameter.

Note:

- The number in tensor minval must be strictly less than maxval at any position after broadcasting.
- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) – The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) – The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (Union[tuple, Tensor]) - The shape of random tensor to be generated. Only constant value is allowed.
- **minval** (Tensor) - The distribution parameter, a. It defines the minimum possibly generated value, with int32 data type. Only one number is supported.
- **maxval** (Tensor) - The distribution parameter, b. It defines the maximum possibly generated value, with int32 data type. Only one number is supported.

Outputs:

Tensor. The shape is the same as the input 'shape', and the data type is int32.

Raises

- **TypeError** – If neither *seed* nor *seed2* is an int.
- **TypeError** – If *shape* is neither a tuple nor a Tensor.
- **TypeError** – If neither *minval* nor *maxval* is a Tensor.
- **ValueError** – If *shape* is not a constant value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> shape = (2, 4)
>>> minval = Tensor(1, mstype.int32)
>>> maxval = Tensor(5, mstype.int32)
>>> uniform_int = ops.UniformInt(seed=10)
>>> output = uniform_int(shape, minval, maxval)
>>> result = output.shape
>>> print(result)
(2, 4)
```

mindspore.ops.UniformReal

class mindspore.ops.**UniformReal**(*seed=0*, *seed2=0*)

Produces random floating-point values, uniformly distributed to the interval [0, 1).

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.
- Currently, on the Ascend platform, *shape* as a Tensor is not supported. This is supported on CPU/GPU platforms. When the input is a Tensor, the supported data types are as follows:
  - GPU: int32, int64.
  - CPU: int16, int32, int64.

Warning: The Ascend backend does not support the reproducibility of random numbers, so the *seed* and *seed2* parameter have no effect.

Parameters

- **seed** (*int, optional*) – The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) – The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **shape** (Union[tuple, Tensor]) - The shape of tensor to be generated. Only constant value is allowed.

Outputs:

Tensor. The shape that the input 'shape' denotes. The dtype is float32.

Raises

- **TypeError** – If *seed* or *seed2* is not an int.

- **TypeError** – If *shape* is neither a tuple nor a Tensor.
- **ValueError** – If *shape* is not a constant value.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> shape = (2, 2)
>>> uniformreal = ops.UniformReal(seed=2)
>>> output = uniformreal(shape)
>>> result = output.shape
>>> print(result)
(2, 2)
```

18.4.3 Array Operation

API Name	Description	Supported Platforms
mindspore.ops.AffineGrid	Creates a 2D or 3D flow field (sampling grid) based on a batch of affine matrices *theta*.	`Ascend` `GPU` `CPU`
mindspore.ops.BatchToSpace	Divides batch dimension with blocks and interleaves these blocks back into spatial dimensions.	`Ascend` `GPU`
mindspore.ops.BatchToSpaceND	Please use batch_to_space_nd instead.	Deprecated
mindspore.ops.BroadcastTo	Refer to *mindspore.ops.broadcast_to()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.Cast	Returns a tensor with the new specified data type.	`Ascend` `GPU` `CPU`
mindspore.ops.ChannelShuffle	Divide the channels in a tensor of shape $(*, C, H, W)$ into g group and rearrange them as $(*, \frac{C}{g}, g, H * W)$, while retaining the original tensor shape in the final output.	`Ascend` `CPU`
mindspore.ops.Col2Im	Rearranges a row vector to an image.	`Ascend` `GPU` `CPU`
mindspore.ops.Concat	Refer to *mindspore.ops.cat()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.Cummax	Refer to *mindspore.ops.cummax()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.Cummin	Returns the cumulative minimum of elements and the index.	`Ascend` `GPU` `CPU`
mindspore.ops.CumProd	Computes the cumulative product of the tensor x along axis.	`Ascend` `GPU` `CPU`
mindspore.ops.CumSum	Computes the cumulative sum of input tensor along axis.	`Ascend` `GPU` `CPU`
mindspore.ops.DataFormatDimMap	Returns the dimension index in the destination data format given in the source data format.	`Ascend` `GPU` `CPU`
mindspore.ops.DepthToSpace	Rearrange blocks of depth data into spatial dimensions.	`Ascend` `GPU` `CPU`
mindspore.ops.Diag	Refer to *mindspore.ops.diag()* for more details.	`Ascend` `GPU` `CPU`

continues on next page

Table 17 – continued from previous page

mindspore.ops.DType	Returns the data type of the input tensor as mindspore.dtype.	Ascend GPU CPU
mindspore.ops.ExpandDims	Refer to mindspore.ops.expand_dims() for more details.	Ascend GPU CPU
mindspore.ops.FillDiagonal	Fills the main diagonal of a Tensor in-place with a specified value and returns the result.	Ascend GPU CPU
mindspore.ops.FillV2	Creates a tensor with shape described by *shape* and fills it with values in *value* .	Ascend GPU CPU
mindspore.ops.FloatStatus	Determines if the elements contain Not a Number(NaN), infinite or negative infinite.	GPU
mindspore.ops.Fmax	Computes the maximum of input tensors element-wise.	CPU
mindspore.ops.Gather	Refer to mindspore.ops.gather() for more details.	Ascend GPU CPU
mindspore.ops.GatherD	Refer to mindspore.ops.gather_d() for more details.	Ascend GPU CPU
mindspore.ops.GatherNd	Refer to mindspore.ops.gather_nd() for more details.	Ascend GPU CPU
mindspore.ops.HammingWindow	Computes the hamming window function with input window length.	Ascend GPU CPU
mindspore.ops.Heaviside	Applies the Heaviside step function for input *x* element-wise.	Ascend GPU CPU
mindspore.ops.HistogramFixedWidth	Returns a rank 1 histogram counting the number of entries in values that fall into every bin.	Ascend GPU
mindspore.ops.Hypot	Computes hypotenuse of input tensors element-wise as legs of a right triangle.	Ascend GPU CPU
mindspore.ops.Identity	Refer to mindspore.ops.deepcopy() for more details.	Ascend GPU CPU
mindspore.ops.Igamma	Calculates lower regularized incomplete Gamma function.	Ascend GPU CPU
mindspore.ops.Igammac	Compute the upper regularized incomplete Gamma function Q(a, x).	Ascend GPU CPU
mindspore.ops.Im2Col	Extracts sliding local blocks from a batched input tensor.	Ascend GPU CPU
mindspore.ops.IndexAdd	Adds tensor *y* to specified axis and indices of tensor *x*.	Ascend GPU CPU
mindspore.ops.IndexFill	Fills the elements under the *dim* dimension of the input Tensor *x* with the input *value* by selecting the indices in the order given in *index*.	Ascend GPU CPU
mindspore.ops.IndexPut	According to the index number of *indexes*, replace the value corresponding to *x1* with the value in *x2*.	Ascend CPU
mindspore.ops.InplaceAdd	Adds *v* into specified rows of *x*.	Ascend GPU CPU
mindspore.ops.InplaceIndexAdd	Adds Tensor *updates* to specified axis and indices of Tensor *var* element-wise.	Ascend CPU
mindspore.ops.InplaceSub	Subtracts *v* into specified rows of *x*.	Ascend GPU CPU
mindspore.ops.InplaceUpdate	Please use mindspore.ops.InplaceUpdateV2 instead.	Deprecated
mindspore.ops.InplaceUpdateV2	Updates specified values in *x* to *v* according to *indices*.	GPU CPU
mindspore.ops.InvertPermutation	Computes the inverse of an index permutation.	Ascend GPU CPU

continues on next page

Table 17 – continued from previous page

mindspore.ops.IsClose	Returns a tensor of Boolean values indicating whether each element of *input* is "close" to the corresponding element of *other*.	Ascend GPU CPU
mindspore.ops.Lcm	Computes least common multiplier of input tensors element-wise.	Ascend GPU CPU
mindspore.ops.LeftShift	Shift the value of each position of the tensor to the left several bits.	Ascend GPU CPU
mindspore.ops.LogSpace	Generates a 1-D Tensor with a length of steps.	Ascend GPU CPU
mindspore.ops.LuUnpack	Converts *LU_data* and *LU_pivots* back into P, L and U matrices, where P is a permutation matrix, L is a lower triangular matrix, and U is an upper triangular matrix.	GPU CPU
mindspore.ops.MaskedFill	Refer to mindspore.ops.masked_fill() for more details.	Ascend GPU CPU
mindspore.ops.MaskedScatter	Updates the value in the input with value in *updates* according to the *mask*.	Ascend CPU
mindspore.ops.MaskedSelect	Refer to mindspore.ops.masked_select() for more details.	Ascend GPU CPU
mindspore.ops.MatrixBandPart	Extracts the central diagonal band of each matrix in a tensor, with all values outside the central band set to zero.	Ascend GPU CPU
mindspore.ops.MatrixDiagPartV3	Returns the diagonal part of a tensor.	Ascend GPU CPU
mindspore.ops.MatrixDiagV3	Constructs a diagonal matrix or a batch of diagonal matrices from a given input Tensor.	Ascend GPU CPU
mindspore.ops.MatrixSetDiagV3	Updates the diagonal part of a batched tensor.	Ascend GPU CPU
mindspore.ops.MatrixSolve	Solves systems of linear equations.	Ascend CPU
mindspore.ops.Meshgrid	Generates coordinate matrices from given coordinate tensors.	Ascend GPU CPU
mindspore.ops.Mvlgamma	Calculates the multivariate log-gamma function element-wise for a given dimension *p*.	Ascend GPU CPU
mindspore.ops.NanToNum	Refer to mindspore.ops.nan_to_num() for more details.	Ascend CPU
mindspore.ops.NonZero	Return a Tensor of the positions of all non-zero values.	Ascend GPU CPU
mindspore.ops.ParallelConcat	Concats input tensors along the first dimension.	Ascend GPU CPU
mindspore.ops.PopulationCount	Computes element-wise population count(a.k.a bitsum, bitcount).	Ascend GPU CPU
mindspore.ops.RandomShuffle	Randomly shuffles a Tensor along its first dimension.	Ascend GPU CPU
mindspore.ops.Range	Refer to mindspore.ops.range() for more details.	GPU CPU
mindspore.ops.Rank	Returns the rank of a tensor.	Ascend GPU CPU
mindspore.ops.Renorm	Renormalizes the sub-tensors along dimension *dim*, and each sub-tensor's p-norm should not exceed the 'maxnorm'.	Ascend GPU CPU
mindspore.ops.Reshape	Refer to mindspore.ops.reshape() for more details.	Ascend GPU CPU
mindspore.ops.ReverseSequence	Reverses variable length slices.	Ascend GPU CPU
mindspore.ops.ReverseV2	Refer to mindspore.ops.flip() for more details.	Ascend GPU CPU

continues on next page

Table 17 – continued from previous page

mindspore.ops.RightShift	Shift the value of each position of Tensor *input_x* to the right by corresponding bits in Tensor *input_y*.	Ascend GPU CPU
mindspore.ops.ScatterNd	Refer to mindspore.ops.scatter_nd() for more details.	Ascend GPU CPU
mindspore.ops.ScatterNdDiv	Applies sparse division to individual values or slices in a tensor.	GPU CPU
mindspore.ops.ScatterNdMax	Computes sparse maximum to individual values or slices in a tensor.	Ascend GPU CPU
mindspore.ops.ScatterNdMin	Applies sparse minimum to individual values or slices in a tensor.	Ascend GPU CPU
mindspore.ops.ScatterNdMul	Applies sparse multiplication to individual values or slices in a tensor.	GPU CPU
mindspore.ops.SearchSorted	Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.	Ascend GPU CPU
mindspore.ops.Select	Refer to mindspore.ops.select() for more details.	Ascend GPU CPU
mindspore.ops.Shape	Returns the shape of the input tensor.	Ascend GPU CPU
mindspore.ops.Size	Returns a Scalar of type int that represents the size of the input Tensor and the total number of elements in the Tensor.	Ascend GPU CPU
mindspore.ops.Slice	Slices a tensor in the specified shape.	Ascend GPU CPU
mindspore.ops.Sort	Sorts the elements of the input tensor along the given dimension in the specified order.	Ascend GPU CPU
mindspore.ops.SpaceToBatchND	Divides spatial dimensions into blocks and combines the block size with the original batch.	Ascend GPU CPU
mindspore.ops.SpaceToDepth	Rearrange blocks of spatial data into depth.	Ascend GPU CPU
mindspore.ops.SparseGatherV2	Returns a slice of input tensor based on the specified indices and axis.	Ascend GPU
mindspore.ops.Split	Splits the input tensor into output_num of tensors along the given axis and output numbers.	Ascend GPU CPU
mindspore.ops.Squeeze	Return the Tensor after deleting the dimension of size 1 in the specified *axis*.	Ascend GPU CPU
mindspore.ops.Stack	Stacks a list of tensors in specified axis.	Ascend GPU CPU
mindspore.ops.StridedSlice	Refer to mindspore.ops.strided_slice() for more details.	Ascend GPU CPU
mindspore.ops.TensorScatterAdd	Creates a new tensor by adding the values from the positions in *input_x* indicated by *indices*, with values from *updates*.	Ascend GPU CPU
mindspore.ops.TensorScatterDiv	Creates a new tensor by dividing the values from the positions in *input_x* indicated by *indices*, with values from *updates*.	GPU CPU
mindspore.ops.TensorScatterMax	By comparing the value at the position indicated by *indices* in *x* with the value in the *updates*, the value at the index will eventually be equal to the largest one to create a new tensor.	GPU CPU
mindspore.ops.TensorScatterMin	By comparing the value at the position indicated by *indices* in *input_x* with the value in the *updates*, the value at the index will eventually be equal to the smallest one to create a new tensor.	Ascend GPU CPU

continues on next page

Table 17 – continued from previous page

mindspore.ops.TensorScatterMul	Creates a new tensor by multiplying the values from the positions in *input_x* indicated by *indices*, with values from *updates*.	GPU CPU
mindspore.ops.TensorScatterSub	Creates a new tensor by subtracting the values from the positions in *input_x* indicated by *indices*, with values from *updates*.	Ascend GPU CPU
mindspore.ops.TensorScatterUpdate	Creates a new tensor by updating the positions in *input_x* indicated by *indices*, with values from *update*.	Ascend GPU CPU
mindspore.ops.TensorShape	Returns the shape of the input tensor.	Ascend GPU CPU
mindspore.ops.Tile	Replicates an input tensor with given multiple times.	Ascend GPU CPU
mindspore.ops.Trace	Refer to mindspore.ops.trace() for more details.	Ascend GPU CPU
mindspore.ops.Transpose	Refer to mindspore.ops.transpose() for more details.	Ascend GPU CPU
mindspore.ops.Tril	Returns the lower triangular portion of the 2-D matrix or the set of matrices in a batch.	Ascend GPU CPU
mindspore.ops.TrilIndices	Computes the indices of the lower triangular elements of a 2D matrix and returns them as a Tensor.	Ascend GPU CPU
mindspore.ops.Triu	Refer to mindspore.ops.triu() for more details.	Ascend GPU CPU
mindspore.ops.TriuIndices	Calculates the indices of the upper triangular elements in a *row* * *col* matrix and returns them as a 2-by-N Tensor.	Ascend GPU CPU
mindspore.ops.Unique	Returns the unique elements of input tensor and also return a tensor containing the index of each value of input tensor corresponding to the output unique tensor.	Ascend GPU CPU
mindspore.ops.UniqueConsecutive	Returns the elements that are unique in each consecutive group of equivalent elements in the input tensor.	Ascend GPU CPU
mindspore.ops.UniqueWithPad	'ops.UniqueWithPad' is deprecated from version 2.4 and will be removed in a future version.	Deprecated
mindspore.ops.UnsortedSegmentMax	Computes the maximum along segments of a tensor.	Ascend GPU CPU
mindspore.ops.UnsortedSegmentMin	Computes the minimum of a tensor along segments.	Ascend GPU CPU
mindspore.ops.UnsortedSegmentProd	Computes the product of a tensor along segments.	Ascend GPU CPU
mindspore.ops.UnsortedSegmentSum	Refer to mindspore.ops.unsorted_segment_sum() for more details.	Ascend GPU CPU
mindspore.ops.Unstack	Unstacks tensor in specified axis, this is the opposite of ops.Stack.	Ascend GPU CPU

mindspore.ops.AffineGrid

class mindspore.ops.**AffineGrid**(*align_corners=False*)

Creates a 2D or 3D flow field (sampling grid) based on a batch of affine matrices *theta*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.affine_grid()* for more details.

Parameters

align_corners (*bool, optional*) – Geometrically, each pixel of input is viewed as a squqre instead of dot. If `True`, consider extremum -1 and 1 referring to the centers of the pixels rather than pixel corners. The default value is `False` , extremum -1 and 1 refer to the corners of the pixels, so that sampling is irrelevant to resolution of the image. Default: `False` .

Inputs:

- **theta** (Tensor) - The input tensor of flow field whose dtype is float16, float32. Input batch of affine matrices with shape $(N, 2, 3)$ for 2D grid or $(N, 3, 4)$ for 3D grid.
- **output_size** (tuple[int]) - The target output image size. The value of target output with format (N, C, H, W) for 2D grid or (N, C, D, H, W) for 3D grid.

Outputs:

Tensor, a tensor whose data type is same as *theta*, and the shape is $(N, H, W, 2)$ for 2D grid or $(N, D, H, W, 3)$ for 3D grid.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> affinegrid = ops.AffineGrid(align_corners=False)
>>> theta = Tensor([[[0.8, 0.5, 0],[-0.5, 0.8, 0]]], mindspore.float32)
>>> out_size = (1, 3, 2, 3)
>>> output = affinegrid(theta, out_size)
>>> print(output)
[[[[-0.78333336 -0.06666666]
[-0.25       -0.4       ]
[ 0.28333336 -0.73333335]]
[[-0.28333336  0.73333335]
[ 0.25        0.4       ]
[ 0.78333336  0.06666666]]]]
```

mindspore.ops.BatchToSpace

class mindspore.ops.**BatchToSpace**(*block_size*, *crops*)

Divides batch dimension with blocks and interleaves these blocks back into spatial dimensions.

This operation will divide batch dimension N into blocks with block_size, the output tensor's N dimension is the corresponding number of blocks after division. The output tensor's H, W dimension is product of original H, W dimension and block_size with given amount to crop from dimension, respectively.

Parameters

- **block_size** (*int*) –The block size of division, has the value not less than 2.
- **crops** (*Union[list(int), tuple(int)]*) –The crop value for H and W dimension, containing 2 subtraction lists. Each list contains 2 integers. All values must be not less than 0. crops[i] specifies the crop values for the spatial dimension i, which corresponds to the input dimension i+2. It is required that $input\_shape[i+2] * block\_size > crops[i][0] + crops[i][1]$.

Inputs:

- **input_x** (Tensor) - The input tensor. It must be a 4-D tensor, dimension 0 must be divisible by product of *block_shape*. The data type is float16 or float32.

Outputs:

Tensor, the output tensor with the same type as input. Assume input shape is (n, c, h, w) with block_size and crops. The output shape will be (n', c', h', w'), where

$n' = n//(block\_size * block\_size)$

$c' = c$

$h' = h * block\_size - crops[0][0] - crops[0][1]$

$w' = w * block\_size - crops[1][0] - crops[1][1]$

Raises

- **TypeError** –If *block_size* or element of *crops* is not an int.
- **TypeError** –If *crops* is neither list nor tuple.
- **ValueError** –If *block_size* is less than 2.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> block_size = 2
>>> crops = [[0, 0], [0, 0]]
>>> batch_to_space = ops.BatchToSpace(block_size, crops)
>>> input_x = Tensor(np.array([[[[1]]], [[[2]]], [[[3]]], [[[4]]]]), mindspore.float32)
>>> output = batch_to_space(input_x)
>>> print(output)
[[[[1.  2.]
   [3.  4.]]]]
```

mindspore.ops.BatchToSpaceND

class mindspore.ops.**BatchToSpaceND**(*block_shape*, *crops*)

Please use batch_to_space_nd instead.

Supported Platforms:

Deprecated

mindspore.ops.BroadcastTo

class mindspore.ops.**BroadcastTo**(*shape*)

```
prim = ops.BroadcastTo(shape)
out = prim(input)
```

is equivalent to

```
ops.broadcast_to(input, shape)
```

Refer to *mindspore.ops.broadcast_to()* for more details.

mindspore.ops.Cast

class mindspore.ops.**Cast**

Returns a tensor with the new specified data type.

Note: When converting complex numbers to boolean type, the imaginary part of the complex number is not taken into account. As long as the real part is non-zero, it returns True; otherwise, it returns False.

Inputs:

- **input** (Union[Tensor, Number]) - The shape of tensor is $(x_1, x_2, ..., x_R)$. The tensor to be cast.
- **dtype** (dtype.Number) - The valid data type of the output tensor. Only constant value is allowed.

Outputs:

Tensor, the shape of tensor is the same as *input*, $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** – If *input* is neither Tensor nor Number.
- **TypeError** – If *dtype* is not a Number.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_np = np.random.randn(2, 3, 4, 5).astype(np.float32)
>>> input = Tensor(input_np)
>>> dtype = mindspore.int32
>>> cast = ops.Cast()
>>> output = cast(input, dtype)
>>> print(output.dtype)
Int32
>>> print(output.shape)
(2, 3, 4, 5)
```

mindspore.ops.ChannelShuffle

class mindspore.ops.**ChannelShuffle**(*group*)

Divide the channels in a tensor of shape $(*, C, H, W)$ into g group and rearrange them as $(*, \frac{C}{g}, g, H * W)$, while retaining the original tensor shape in the final output.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.channel_shuffle()* for more detail.

Parameters

group (*int*) –Number of group to divide channels in.

Inputs:

- **x** (Tensor) - Tensor to be divided, it has shape $(*, C, H, W)$, with float16, float32, int8, int16, int32, int64, uint8, uint16, uint32, uint64 data type.

Outputs:

A Tensor, has the same type as the x, and has the shape $(*, C, H, W)$.

Supported Platforms:

Ascend CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> group = 2
>>> x = Tensor(np.arange(1 * 4 * 2 * 2).reshape(1, 4, 2, 2).astype(np.int16))
>>> channel_shuffle_func = ops.ChannelShuffle(group)
>>> y = channel_shuffle_func(x)
>>> print(y)
[[[[ 0  1]
   [ 2  3]]
   [[ 8  9]
```

(continues on next page)

(continued from previous page)

```
[10 11]]
[[ 4  5]
[ 6  7]]
[[12 13]
[14 15]]]]
```

mindspore.ops.Col2Im

class mindspore.ops.**Col2Im**(*kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Rearranges a row vector to an image. It is usually used to reconstruct an image from a set of image patches(or sliding local blocks).

Consider an input Tensor of shape $(N, C, \prod(\text{kernel\_size}), L)$, where N is batch dimension, C is channel dimension, $\prod(\text{kernel\_size})$ is the block size, and L is the total number of blocks. This operation combines these local blocks into the large `output` tensor of shape $(N, C, \text{output\_size}[0], \text{output\_size}[1], \dots)$ by summing the overlapping values.

$$L = \prod_{d} \left\lfloor \frac{\text{output\_size}[d] + 2 \times \text{padding}[d] - \text{dilation}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{stride}[d]} + 1 \right\rfloor$$

where d is over all spatial dimensions. The *padding*, *stride* and *dilation* arguments specify how the sliding blocks are retrieved.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **kernel_size** (*Union[int, tuple[int], list[int]]*) – The size of the kernel, should be two positive int for height and width. If type is int, it means that height equal with width. Must be specified.
- **dilation** (*Union[int, tuple[int], list[int]], optional*) – The size of the dilation, should be two positive int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **padding** (*Union[int, tuple[int], list[int]], optional*) – The size of the padding, should be two int for height and width. If type is int, it means that height equal with width. Default: `0` .
- **stride** (*Union[int, tuple[int], list[int]], optional*) – The size of the stride, should be two positive int for height and width. If type is int, it means that height equal with width. Default: `1` .

Inputs:

- **x** (Tensor) - 4D input Tensor.
- **output_size** (Tensor) - 1D tensor with 2 elements of data type int32 or int64.

Outputs:

Tensor, a 4-D Tensor with same type of input *x*.

Raises

- **TypeError** – If dtype of *kernel_size* , *dilation* , *padding* or *stride* is not in Union[int, tuple[int], list[int]].
- **ValueError** – If values in *kernel_size* , *dilation* , *padding* or *stride* are not greater than zero or any one of them has more than 2 elements.
- **ValueError** – If x.shape[2] != kernel_size[0] * kernel_size[1].
- **ValueError** – If x.shape[3] does not match the calculated number of sliding blocks.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(input_data=np.random.rand(16, 16, 4, 25), dtype=mstype.float32)
>>> output_size = Tensor(input_data=[8, 8], dtype=mstype.int32)
>>> col2im = ops.Col2Im(kernel_size=[2, 2], dilation=[2, 2], padding=[2, 2], stride=[2, 2])
>>> y = col2im(x, output_size)
>>> print(y.shape)
(16, 16, 8, 8)
```

mindspore.ops.Concat

class mindspore.ops.**Concat**(*axis=0*)

```
prim = ops.Concat(axis)
out = prim(tensors)
```

is equivalent to

```
ops.cat(tensors, axis)
```

Refer to *mindspore.ops.cat()* for more details.

mindspore.ops.Cummax

class mindspore.ops.**Cummax**(*axis*)

```
prim = ops.Cummax(axis)
out = prim(input)
```

is equivalent to

```
ops.cummax(input, axis)
```

Refer to *mindspore.ops.cummax()* for more details.

mindspore.ops.Cummin

class mindspore.ops.**Cummin**(*axis*)

Returns the cumulative minimum of elements and the index.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.cummin()* for more detail.

Parameters

axis (*int*) –The axis to accumulate the tensor's value. Must be in the range [-rank(input), rank(input)).

Inputs:

- **input** (Tensor) - The input tensor.

Outputs:

A tuple of 2 Tensors(values, indices), containing the cumulative minimum of elements and the index, the shape of each output tensor is the same as input *input*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> import mindspore
>>> a = Tensor([-0.2284, -0.6628,  0.0975,  0.2680, -1.3298, -0.4220], mindspore.float32)
>>> axis = 0
>>> output = ops.Cummin(axis)(a)
>>> print(output[0])
[-0.2284 -0.6628 -0.6628 -0.6628 -1.3298 -1.3298]
>>> print(output[1])
[0 1 1 1 4 4]
```

mindspore.ops.CumProd

class mindspore.ops.**CumProd**(*exclusive=False*, *reverse=False*)

Computes the cumulative product of the tensor x along axis. For example, if input is a vector of size N, the result will also be a vector of size N, with elements.

$$y_i = x_1 * x_2 * x_3 * ... * x_i$$

Parameters

- **exclusive** (*bool, optional*) –If `True` , perform exclusive cumulative product. Default: `False` .
- **reverse** (*bool, optional*) –If `True` , reverse the result along axis. Default: `False` .

Inputs:

- **x** (Tensor[Number]) - The input Tensor with shape $(N, *)$ where $*$ means any number of additional dimensions.
- **axis** (int) - The dimensions to compute the cumulative product. Only constant value is allowed.

Outputs:

Tensor, has the same shape and dtype as the *x*.

Raises

- **TypeError** –If *exclusive* or *reverse* is not a bool.
- **TypeError** –If *axis* is not an int.
- **ValueError** –If *axis* is None.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> a, b, c, = 1, 2, 3
>>> x = Tensor(np.array([a, b, c]).astype(np.float32))
>>> op0 = ops.CumProd()
>>> output0 = op0(x, 0) # output=[a, a * b, a * b * c]
>>> op1 = ops.CumProd(exclusive=True)
>>> output1 = op1(x, 0) # output=[1, a, a * b]
>>> op2 = ops.CumProd(reverse=True)
>>> output2 = op2(x, 0) # output=[a * b * c, b * c, c]
>>> op3 = ops.CumProd(exclusive=True, reverse=True)
>>> output3 = op3(x, 0) # output=[b * c, c, 1]
>>> print(output0)
[1. 2. 6.]
>>> print(output1)
[1. 1. 2.]
>>> print(output2)
[6. 6. 3.]
>>> print(output3)
[6. 3. 1.]
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [5, 3, 5]]).astype(np.float32))
>>> output4 = op0(x, 0)
>>> output5 = op0(x, 1)
>>> print(output4)
[[ 1.  2.  3.]
 [ 4. 10. 18.]
 [20. 30. 90.]]
>>> print(output5)
[[  1.   2.   6.]
 [  4.  20. 120.]
 [  5.  15.  75.]]
```

mindspore.ops.CumSum

class mindspore.ops.**CumSum**(*exclusive=False*, *reverse=False*)

Computes the cumulative sum of input tensor along axis.

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Parameters

- **exclusive** (*bool, optional*) – By default, this op performs an inclusive cumsum, which means that the first element of the input is identical to the first element of the output. Default: `False` .
- **reverse** (*bool, optional*) – If `True` , perform inverse cumulative sum. Default: `False` .

Inputs:

- **input** (Tensor) - The input Tensor with shape $(N, *)$ where $*$ means any number of additional dimensions.
- **axis** (int) - The axis to accumulate the tensor's value. Only constant value is allowed. Must be in the range [-rank(input), rank(input)).

Outputs:

Tensor, the shape of the output tensor is consistent with the input tensor's.

Raises

- **TypeError** –If *exclusive* or *reverse* is not a bool.
- **TypeError** –If *axis* is not an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[3, 4, 6, 10], [1, 6, 7, 9], [4, 3, 8, 7], [1, 3, 7, 9]]).astype(np.
↪float32))
>>> cumsum = ops.CumSum()
>>> # case 1: along the axis 0
>>> y = cumsum(x, 0)
>>> print(y)
[[ 3.  4.  6. 10.]
 [ 4. 10. 13. 19.]
 [ 8. 13. 21. 26.]
 [ 9. 16. 28. 35.]]
>>> # case 2: along the axis 1
>>> y = cumsum(x, 1)
>>> print(y)
[[ 3.  7. 13. 23.]
 [ 1.  7. 14. 23.]
 [ 4.  7. 15. 22.]
 [ 1.  4. 11. 20.]]
>>> # Next demonstrate exclusive and reverse, along axis 1
>>> # case 3: exclusive = True
>>> cumsum = ops.CumSum(exclusive=True)
>>> y = cumsum(x, 1)
>>> print(y)
[[ 0.  3.  7. 13.]
 [ 0.  1.  7. 14.]
 [ 0.  4.  7. 15.]
 [ 0.  1.  4. 11.]]
>>> # case 4: reverse = True
>>> cumsum = ops.CumSum(reverse=True)
>>> y = cumsum(x, 1)
>>> print(y)
[[23. 20. 16. 10.]
 [23. 22. 16.  9.]
 [22. 18. 15.  7.]
 [20. 19. 16.  9.]]
```

mindspore.ops.DataFormatDimMap

class mindspore.ops.**DataFormatDimMap**(*src_format='NHWC'*, *dst_format='NCHW'*)

Returns the dimension index in the destination data format given in the source data format.

Parameters

- **src_format** (*str*) – An optional value for source data format. The format can be `'NHWC'` and `'NCHW'`. Default: `'NHWC'`.
- **dst_format** (*str*) – An optional value for destination data format. The format can be `'NHWC'` and `'NCHW'`. Default: `'NCHW'`.

Inputs:

- **input_x** (Tensor) - A Tensor, each element is used as a dimension index of the source data format. The suggested values are in the range [-4, 4). Only supports int32.

Outputs:

Tensor, Return the dimension index in the given target data format, has the same data type and shape as the *input_x*.

Raises

- **TypeError** – If *src_format* or *dst_format* is not a str.
- **TypeError** – If *input_x* is not a Tensor whose dtype is not int32.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input_x = Tensor([0, 1, 2, 3], mindspore.int32)
>>> dfdm = ops.DataFormatDimMap()
>>> output = dfdm(input_x)
>>> print(output)
[0 3 1 2]
```

mindspore.ops.DepthToSpace

class mindspore.ops.**DepthToSpace**(*block_size*)

Rearrange blocks of depth data into spatial dimensions.

This is the reverse operation of SpaceToDepth.

The depth of output tensor is $input\_depth/(block\_size * block\_size)$.

The output tensor's *height* dimension is $height * block\_size$.

The output tensor's *weight* dimension is $weight * block\_size$.

The input tensor's depth must be divisible by *block_size* * *block_size*. The data format is "NCHW".

Parameters

block_size (*int*) – The block size used to divide depth data. It must be >= 2.

Inputs:

- **x** (Tensor) - The target tensor. It must be a 4-D tensor with shape $(N, C_{in}, H_{in}, W_{in})$. The data type is Number.

Outputs:

Tensor of shape $(N, C_{in}/\text{block\_size}^2, H_{in} * \text{block\_size}, W_{in} * \text{block\_size})$.

Raises

- **TypeError** – If *block_size* is not an int.
- **ValueError** – If *block_size* is less than 2.
- **ValueError** – If length of shape of *x* is not equal to 4.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.rand(1, 12, 1, 1), mindspore.float32)
>>> block_size = 2
>>> depth_to_space = ops.DepthToSpace(block_size)
>>> output = depth_to_space(x)
>>> print(output.shape)
(1, 3, 2, 2)
```

mindspore.ops.Diag

class mindspore.ops.**Diag**

```
prim = ops.Diag()
out = prim(input)
```

is equivalent to

```
ops.diag(input)
```

Refer to *mindspore.ops.diag()* for more details.

mindspore.ops.DType

class mindspore.ops.**DType**

Returns the data type of the input tensor as mindspore.dtype.

Inputs:

- **input_x** (Tensor) - Input Tensor.

Outputs:

mindspore.dtype, the data type of a tensor.

Raises

TypeError – If *input_x* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_tensor = Tensor(np.array([[2, 2], [2, 2]]), mindspore.float32)
>>> output = ops.DType()(input_tensor)
>>> print(output)
Float32
```

mindspore.ops.ExpandDims

class mindspore.ops.**ExpandDims**

```
prim = ops.ExpandDims()
out = prim(input_x, axis)
```

is equivalent to

```
ops.expand_dims(input_x, axis)
```

Refer to *mindspore.ops.expand_dims()* for more details.

mindspore.ops.FillDiagonal

class mindspore.ops.**FillDiagonal**(*fill_value*, *wrap=False*)

Fills the main diagonal of a Tensor in-place with a specified value and returns the result. The input has at least 2 dimensions, and all dimensions of input must be equal in length when the dimension of input is greater than 2.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

- **fill_value** (*float*) – The value to fill the diagonal of *input_x*.
- **wrap** (*bool, optional*) – Controls whether the diagonal elements continue onto the remaining rows in case of a tall matrix(A matrix has more rows than columns). Examples blow demonstrates how it works on a tall matrix if *wrap* is set `True` . Default: `False` .

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

- **y** (Tensor) - Tensor, has the same shape and data type as the input *input_x*.

Raises

- **ValueError** –If the dimension of *input_x* is not greater than 1.
- **ValueError** –If the size of each dimension is not equal, when the dimension is greater than 2.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(np.float32))
>>> fill_value = 9.9
>>> fill_diagonal = ops.FillDiagonal(fill_value)
>>> y = fill_diagonal(x)
>>> print(y)
[[9.9 2.  3. ]
 [4.  9.9 6. ]
 [7.  8.  9.9]]
>>> x = Tensor(np.array([[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4], [5, 5, 5]]).
↪astype(np.int32))
>>> fill_value = 9.0
>>> fill_diagonal = ops.FillDiagonal(fill_value)
>>> y = fill_diagonal(x)
>>> print(y)
[[9 0 0]
 [1 9 1]
 [2 2 9]
 [3 3 3]
 [4 4 4]
 [5 5 5]]
>>> x = Tensor(np.array([[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3],
...                      [4, 4, 4], [5, 5, 5], [6, 6, 6]]).astype(np.int64))
>>> fill_value = 9.0
>>> wrap = True
>>> fill_diagonal = FillDiagonal(fill_value, wrap)
>>> y = fill_diagonal(x)
>>> print(y)
[[9 0 0]
 [1 9 1]
 [2 2 9]
 [3 3 3]
 [9 4 4]
 [5 9 5]
 [6 6 9]]
```

mindspore.ops.FillV2

class mindspore.ops.**FillV2**

Creates a tensor with shape described by *shape* and fills it with values in *value* .

Inputs:

- **shape** (Union[Tuple[int], Tensor[int]]) - 1-D Tensor or Tuple, specify the shape of output tensor. Its dtype must be int32 or int64.
- **value** (Tensor) - A 0-D Tensor, the value to fill the output tensor *y* .

Outputs:

- **y** (Tensor) - A tensor, its shape and value are described above.

Raises

- **TypeError** –If *shape* is not a 1-D tensor or tuple.
- **TypeError** –If the data type of *shape* is not int32 or int64.
- **ValueError** –If *value* is not a 0-D Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> fillV2 = ops.FillV2()
>>> output = fillV2(Tensor([2, 3], mindspore.int32), Tensor(1, mindspore.float32))
>>> print(output)
[[1. 1. 1.]
 [1. 1. 1.]]
>>> output = fillV2(Tensor([3, 3], mindspore.int64), Tensor(0, mindspore.int32))
>>> print(output)
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

mindspore.ops.FloatStatus

class mindspore.ops.**FloatStatus**

Determines if the elements contain Not a Number(NaN), infinite or negative infinite. 0 for normal, 1 for overflow.

Inputs:

- **x** (Tensor) - The input tensor. The data type must be float16, float32 or float64. $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tensor, has the shape of $(1,)$, and the dtype is *mindspore.dtype.float32*.

Raises

TypeError – If dtype of *x* is not in [float16, float32, float64].

Supported Platforms:

GPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> float_status = ops.FloatStatus()
>>> x = Tensor(np.array([np.log(-1), 1, np.log(0)]), mindspore.float32)
>>> result = float_status(x)
>>> print(result)
[1.]
```

mindspore.ops.Fmax

class mindspore.ops.**Fmax**

Computes the maximum of input tensors element-wise.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.fmax()* for more detail.

Supported Platforms:

CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([1.0, 5.0, 3.0]), mindspore.float32)
>>> x2 = Tensor(np.array([4.0, 2.0, 6.0]), mindspore.float32)
>>> fmax = ops.Fmax()
>>> output = fmax(x1, x2)
>>> print(output)
[4. 5. 6.]
```

mindspore.ops.Gather

class mindspore.ops.**Gather**(*batch_dims=0*)

```
prim = ops.Gather(batch_dims)
out = prim(input_params, input_indices, axis)
```

is equivalent to

```
ops.gather(input_params, input_indices, axis, batch_dims)
```

Refer to *mindspore.ops.gather()* for more details.

mindspore.ops.GatherD

class mindspore.ops.**GatherD**

```
prim = ops.GatherD()
out = prim(x, dim, index)
```

is equivalent to

```
ops.gather_d(x, dim, index)
```

Refer to *mindspore.ops.gather_d()* for more details.

mindspore.ops.GatherNd

class mindspore.ops.**GatherNd**

```
prim = ops.GatherNd()
out = prim(input_x, indices)
```

is equivalent to

```
ops.gather_nd(input_x, indices)
```

Refer to *mindspore.ops.gather_nd()* for more details.

mindspore.ops.HammingWindow

class mindspore.ops.**HammingWindow**(*periodic=True*, *alpha=0.54*, *beta=0.46*, *dtype=mstype.float32*)

Computes the hamming window function with input window length.

$$w[n] = \alpha - \beta \cos\left(\frac{2\pi n}{N-1}\right),$$

where N is the full window size.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **periodic** (*bool, optional*) –a flag determines whether the returned window trims off the last duplicate value from the symmetric window. Default: `True` .
  - If True, returns a window to be used as periodic function, in above formula, N = length + 1.
  - If False, return a symmetric window, N = length.
- **alpha** (*float, optional*) –The coefficient α in the equation above. Default: `0.54` .
- **beta** (*float, optional*) –The coefficient β in the equation above. Default: `0.46` .
- **dtype** (*mindspore.dtype*, optional) –An optional data type of `mstype.float16` , `mstype.float32` and `mstype.float64` . Default: `mstype.float32`.

Inputs:

- **length** (Tensor) - a positive integer tensor controlling the returned window size, must be 1D.

Outputs:

Tensor, A 1-D tensor containing the window, whose shape is (length,).

Raises

- **TypeError** –If *length* is not a Tensor.
- **TypeError** –If dtype of *length* is not integer data type.
- **TypeError** –If *periodic* is not a bool.
- **TypeError** –If *alpha* is not a float.
- **TypeError** –If *beta* is not a float.
- **TypeError** –If *dtype* is not mindspore.float16, mindspore.float32 or mindspore.float64.
- **ValueError** –If dimension of *length* is not 1.
- **ValueError** –If data of *length* is negative.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> # case 1: periodic=True.
>>> length = Tensor(np.array([6]).astype(np.int32))
>>> hamming_window = ops.HammingWindow(periodic=True)
>>> y = hamming_window(length)
>>> print(y)
[0.08000001 0.31       0.77000004 1.         0.77000004 0.31      ]
>>> # case 2: periodic=False.
>>> length = Tensor(np.array([7]).astype(np.int32))
>>> hamming_window = ops.HammingWindow(periodic=False)
>>> y = hamming_window(length)
>>> print(y)
[0.08000001 0.31       0.77000004 1.         0.77000004 0.31       0.08000001]
```

mindspore.ops.Heaviside

class mindspore.ops.**Heaviside**

Applies the Heaviside step function for input *x* element-wise.

$$\text{heaviside}(x, \text{values}) = \begin{cases} 0, & \text{if } x < 0 \\ \text{values}, & \text{if } x == 0 \\ 1, & \text{if } x > 0 \end{cases}$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - The input tensor. With real number data type.
- **values** (Tensor) - The values to use where *x* is zero. It should be able to broadcast with *x* have the same dtype as *x*.

Outputs:

Tensor, has the same type as *x* and *values*.

Raises

- **TypeError** – If *x* or *values* is not Tensor.
- **TypeError** – If data type *x* and *values* is different.
- **ValueError** – If shape of two inputs are not broadcastable.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([-1.5, 0., 2.]))
>>> values = Tensor(np.array([0.5]))
>>> heaviside = ops.Heaviside()
>>> y = heaviside(x, values)
>>> print(y)
[0.  0.5 1. ]
```

mindspore.ops.HistogramFixedWidth

class mindspore.ops.**HistogramFixedWidth**(*nbins*, *dtype='int32'*)

Returns a rank 1 histogram counting the number of entries in values that fall into every bin. The bins are equal width and determined by the input *range* and the argument *nbins*.

Parameters

- **nbins** (*int*) – The number of histogram bins, the type is a positive integer.
- **dtype** (*str, optional*) – An optional attribute. The dtype must be str. Default: `'int32'`.

Inputs:

- **x** (Tensor) - Numeric Tensor. Must be one of the following types: int32, float32, float16.
- **range** (Tensor) - Must have the same data type as *x*, and the shape is $(2,)$. x <= range[0] will be mapped to histogram[0], x >= range[1] will be mapped to histogram[-1].

Outputs:

1-D Tensor, whose length is the type is *nbins* with dtype of int32.

Raises

- **TypeError** –If *dtype* is not a str or *nbins* is not an int.
- **ValueError** –If *nbins* is less than 1.
- **ValueError** –If *dtype* is not 'int32'.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> x = Tensor([-1.0, 0.0, 1.5, 2.0, 5.0, 15], mindspore.float16)
>>> range_op = Tensor([0.0, 5.0], mindspore.float16)
>>> hist = ops.HistogramFixedWidth(5)
>>> output = hist(x, range_op)
>>> print(output)
[2 1 1 0 2]
```

mindspore.ops.Hypot

class mindspore.ops.**Hypot**

Computes hypotenuse of input tensors element-wise as legs of a right triangle. The shape of two inputs should be broadcastable, and data type of them should be one of: float32, float64.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x1** (Tensor) - The first input tensor.
- **x2** (Tensor) - The second input tensor.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is one with higher precision in the two inputs.

Raises

- **TypeError** –If data type *x1* or *x2* is not float32 or float64.
- **ValueError** –If shape of two inputs are not broadcastable.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([3., 5., 7.]))
>>> x2 = Tensor(np.array([4., 12., 24.]))
>>> hypot_ = ops.Hypot()
>>> y = hypot_(x1, x2)
>>> print(y)
[ 5. 13. 25.]
>>> x1 = Tensor(2.1, mindspore.float32)
>>> x2 = Tensor(2.1, mindspore.float32)
>>> hypot_ = ops.Hypot()
>>> y = hypot_(x1, x2)
>>> print(y)
2.9698484
```

mindspore.ops.Identity

class mindspore.ops.**Identity**

```
prim = ops.Identity()
out = prim(input_x)
```

is equivalent to

```
ops.deepcopy(input_x)
```

Refer to *mindspore.ops.deepcopy()* for more details.

mindspore.ops.Igamma

class mindspore.ops.**Igamma**

Calculates lower regularized incomplete Gamma function.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.igamma()* for more details.

Inputs:

- **a** (Tensor) - The input tensor.
- **x** (Tensor) - The input tensor. It should have the same dtype with *a*.

Outputs:

Tensor, has the same dtype as *a* and *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> a = Tensor(np.array([2.0, 4.0, 6.0, 8.0]).astype(np.float32))
>>> x = Tensor(np.array([2.0, 3.0, 4.0, 5.0]).astype(np.float32))
>>> igamma = ops.Igamma()
>>> output = igamma(a, x)
>>> print (output)
[0.593994  0.35276785  0.21486944  0.13337152]
>>> a = Tensor(2.1, mindspore.float32)
>>> x = Tensor(2.1, mindspore.float32)
>>> igamma = ops.Igamma()
>>> output = igamma(a, x)
>>> print (output)
0.5917439
```

mindspore.ops.Igammac

class mindspore.ops.**Igammac**

Compute the upper regularized incomplete Gamma function Q(a, x).

Refer to *mindspore.ops.igammac()* for more details.

Inputs:

- **a** (Tensor) - The input tensor.
- **x** (Tensor) - The input tensor. It should have the same dtype with *a*.

Outputs:

Tensor, has the same dtype as *a* and *x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> a = Tensor(np.array([2.0, 4.0, 6.0, 8.0]).astype(np.float32))
>>> x = Tensor(np.array([2.0, 3.0, 4.0, 5.0]).astype(np.float32))
>>> igammac = ops.Igammac()
>>> output = igammac(a, x)
>>> print (output)
[0.40600586 0.6472318  0.7851304  0.8666283 ]
>>> a = Tensor(2.1, mindspore.float32)
>>> x = Tensor(2.1, mindspore.float32)
>>> igammac = ops.Igammac()
>>> output = igammac(a, x)
```

(continues on next page)

(continued from previous page)

```
>>> print (output)
0.40825662
```

mindspore.ops.Im2Col

class mindspore.ops.**Im2Col**(*ksizes*, *strides=1*, *dilations=1*, *pads=0*)

Extracts sliding local blocks from a batched input tensor.

Consider a batched input tensor of shape $(N, C, *)$, where N is the batch dimension, C is the channel dimension, and $*$ represent arbitrary spatial dimensions. This operation flattens each sliding *ksizes*- sized block within the spatial dimensions of input x into a column (i.e., last dimension) of a 4-D output tensor of shape $(N, C, \prod(\text{kernel\_size}), L)$, where $C \times \prod(\text{kernel\_size})$ is the total number of elements within each block (a block has $\prod$(kernel_size) spatial locations each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_d \left\lfloor \frac{\text{spatial\_size}[d] + 2 \times \text{pads}[d] - \text{dilations}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{strides}[d]} + 1 \right\rfloor,$$

where spatial_size is formed by the spatial dimensions of input x ($*$ above), and d is over all spatial dimensions.

Therefore, indexing *output* at the last dimension (column dimension) gives all values within a certain block.

The *pads*, *strides* and *dilations* arguments specify how the sliding blocks are retrieved.

Note: Currently, only 4-D input tensors (batched image-like tensors) are supported.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **ksizes** (*Union[int, tuple[int], list[int]]*) – The size of the kernel, should be two int for height and width. If type is int, it means that height equal with width. Must be specified.
- **strides** (*Union[int, tuple[int], list[int]], optional*) – The stride of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **dilations** (*Union[int, tuple[int], list[int]], optional*) – The dilation of the window, should be two int for height and width. If type is int, it means that height equal with width. Default: `1` .
- **pads** (*Union[int, tuple[int], list[int]], optional*) – The pad of the window, that must be a tuple of one or two *int* for height and width. Default: `0` .
  - If one int, $pad\_height = pad\_width$.
  - If two int, $pad\_height = pads[0]$, $pad\_width = pads[1]$.

Inputs:

- **x** (Tensor) - input tensor, only 4-D input tensors (batched image-like tensors) are supported.

Outputs:

Tensor, a 4-D Tensor with same type of input x.

Raises

- **TypeError** – If *ksizes* data type is not in Union[int, tuple[int], list[int]].

- **TypeError** –If *strides* data type is not in Union[int, tuple[int], list[int]].
- **TypeError** –If *dilations* data type is not in Union[int, tuple[int], list[int]].
- **TypeError** –If *pads* data type isnot in Union[int, tuple[int], list[int]].
- **ValueError** –If *ksizes* value is not greater than zero or elements number more than 2.
- **ValueError** –If *strides* value is not greater than zero or elements number more than 2.
- **ValueError** –If *dilations* value is not greater than zero or elements number more than 2.
- **ValueError** –If *pads* value is not greater than zero.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(input_data=np.random.rand(4, 4, 32, 32), dtype=mstype.float64)
>>> im2col = ops.Im2Col(ksizes=3, strides=1, dilations=1)
>>> y = im2col(x)
>>> print(y.shape)
(4, 4, 9, 900)
```

mindspore.ops.IndexAdd

class mindspore.ops.**IndexAdd**(*axis*, *use_lock=True*, *check_index_bound=True*)

Adds tensor *y* to specified axis and indices of tensor *x*. The axis should be in [-len(x.dim), len(x.dim) - 1], and indices should be in [0, the size of *x* - 1] at the axis dimension.

Parameters

- **axis** (*int*) –The dimension along which to index.
- **use_lock** (*bool, optional*) –Whether to enable a lock to protect the updating process of variable tensors. If `True` , when updating the value of *x*, this process will be protected by a lock by using atomic operation. If `False` , the result may be unpredictable. Default: `True` .
- **check_index_bound** (*bool, optional*) –If `True` , check index boundary. If `False` , don't check index boundary. Default: `True` .

Inputs:

- **x** (Union[Parameter, Tensor]) - The input Parameter or Tensor to add to.
- **indices** (Tensor) - Add the value of *x* and *y* along the dimension of the *axis* according to the specified index value, with data type int32. The *indices* must be 1D with the same size as the size of *y* in the *axis* dimension. The values of *indices* should be in [0, b), where the b is the size of *x* in the *axis* dimension.
- **y** (Tensor) - The input tensor with the value to add. Must have same data type as *x*. The shape must be the same as *x* except the *axis* th dimension.

Outputs:

Tensor, has the same shape and dtype as *x*.

Raises

- **TypeError** –If neither *indices* nor *y* is a Tensor.
- **ValueError** –If axis is out of *x* rank's range.
- **ValueError** –If *x* rank is not the same as *y* rank.
- **ValueError** –If shape of *indices* is not 1D or size of *indices* is not equal to dimension of y[axis].
- **ValueError** –If *y*'s shape is not the same as *x* except the *axis* th dimension.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.index_add = ops.IndexAdd(axis=1)
...         self.x = Parameter(Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.
↪float32),
...                            name="name_x")
...         self.indices = Tensor(np.array([0, 2]), mindspore.int32)
...
...     def construct(self, y):
...         return self.index_add(self.x, self.indices, y)
...
>>> y = Tensor(np.array([[0.5, 1.0], [1.0, 1.5], [2.0, 2.5]]), mindspore.float32)
>>> net = Net()
>>> output = net(y)
>>> print(output)
[[ 1.5  2.   4. ]
 [ 5.   5.   7.5]
 [ 9.   8.  11.5]]
```

mindspore.ops.IndexFill

class mindspore.ops.**IndexFill**

Fills the elements under the *dim* dimension of the input Tensor *x* with the input *value* by selecting the indices in the order given in *index*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.index_fill()* for more details.

Inputs:

- **x** (Tensor) - Input tensor.
- **dim** (Union[int, Tensor]) - Dimension along which to fill the input tensor. Only supports a 0-dimensional tensor or an int number.

- **index** (Tensor) - Indices of the input tensor to fill in.
- **value** (Union[bool, int, float, Tensor]) - Value to fill the input tensor.

Outputs:

Tensor, has the same type and shape as input tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> index_fill = ops.IndexFill()
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(np.float32))
>>> index = Tensor([0, 2], mindspore.int32)
>>> value = Tensor(-2.0, mindspore.float32)
>>> y = index_fill(x, 1, index, value)
>>> print(y)
[[-2. 2. -2.]
 [-2. 5. -2.]
 [-2. 8. -2.]]
```

mindspore.ops.IndexPut

class mindspore.ops.**IndexPut**(*accumulate=0*)

According to the index number of *indexes*, replace the value corresponding to *x1* with the value in *x2*.

Parameters

accumulate (*int*) – If accumulate is 1, the elements in x2 are added to x1, else the elements in x2 replace the corresponding element in x1, should be 0 or 1. Default: `0` .

Inputs:

- **x1** (Tensor) - The assigned target tensor, 1-D or higher dimensional.
- **x2** (Tensor) - 1-D Tensor of the same type as *x1*. If the size of *x2* is 1, it will broadcast to the same size as *x1*.
- **indices** (tuple[Tensor], list[Tensor]) - the indices of type int32 or int64, used to index into x1. The rank of tensors in indices should be 1-D, size of indices should <= x1.rank and the tensors in indices should be broadcastable.

Outputs:

Tensor, has the same dtype and shape as *x1*.

Raises

- **TypeError** – If the dtype of *x1* is not equal to the dtype of *x2*.
- **TypeError** – If *indices* is not tuple[Tensor] or list[Tensor].
- **TypeError** – If the dtype of tensors in *indices* are not int32 or int64.
- **TypeError** – If the dtype of tensors in *indices* are inconsistent.
- **TypeError** – If the dtype of *accumulate* are not int.

- **ValueError** –If rank(x2) is not 1-D.
- **ValueError** –If size(x2) is not 1 or max size of the tensors in *indices* when rank(x1) == size(indices).
- **ValueError** –If size(x2) is not 1 or x1.shape[-1] when rank(x1) > size(indices).
- **ValueError** –If the rank of tensors in *indices* is not 1-D.
- **ValueError** –If the tensors in *indices* is not be broadcastable.
- **ValueError** –If size(indices) > rank(x1).
- **ValueError** –If *accumulate* is not equal to 0 or 1.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([[1, 2, 3], [4, 5, 6]]).astype(np.int32))
>>> x2 = Tensor(np.array([3]).astype(np.int32))
>>> indices = [Tensor(np.array([0, 0]).astype(np.int32)), Tensor(np.array([0, 1]).astype(np.
↪int32))]
>>> accumulate = 1
>>> op = ops.IndexPut(accumulate = accumulate)
>>> output = op(x1, x2, indices)
>>> print(output)
[[4 5 3]
 [4 5 6]]
```

mindspore.ops.InplaceAdd

class mindspore.ops.**InplaceAdd**(*indices*)

Adds *v* into specified rows of *x*. Computes *y* = *x*; y[i,] += *v*.

Refer to *mindspore.ops.inplace_add()* for more details.

Parameters

indices (*Union[int, tuple]*) –Indices into the left-most dimension of *x*, and determines which rows of *x* to add with *v*. It is an integer or a tuple, whose value is in [0, the first dimension size of *x*).

Inputs:

- **x** (Tensor) - The tensor to be added. It has shape $(N, *)$ where $*$ means any number of additional dimensions.
- **input_v** (Tensor) - The value tensor add to *x*. It has the same dimension sizes as *x* except the first dimension, whose size must be the same as *indices*. It has the same data type with *x*.

Outputs:

Tensor, has the same shape and dtype as *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> indices = (0, 1)
>>> x = Tensor(np.array([[1, 2], [3, 4], [5, 6]]), mindspore.float32)
>>> input_v = Tensor(np.array([[0.5, 1.0], [1.0, 1.5]]), mindspore.float32)
>>> inplaceAdd = ops.InplaceAdd(indices)
>>> output = inplaceAdd(x, input_v)
>>> print(output)
[[1.5 3. ]
 [4.  5.5]
 [5.  6. ]]
```

mindspore.ops.InplaceIndexAdd

class mindspore.ops.**InplaceIndexAdd**(*axis*)

Adds Tensor *updates* to specified axis and indices of Tensor *var* element-wise.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.inplace_index_add()* for more details.

Parameters

axis (*int*) – The dimension along which to index. It should be in range $[0, len(var.dim))$.

Inputs:

- **var** (Union[Parameter, Tensor]) - The input Parameter or Tensor to add to, with data type uint8, int8, int16, int32, float16, float32, float64.
- **indices** (Tensor) - The indies along *axis* to perform the addition. A 1D Tensor of shape $(updates.shape[axis],)$, every value of it should be in range $[0, var.shape[axis])$ with data type int32.
- **updates** (Tensor) - The input Tensor with the value to add. Must have same data type as *var*. The shape must be the same as *var* except the *axis* th dimension.

Outputs:

Tensor, updated result, has the same shape and dtype as *var*.

Supported Platforms:

Ascend CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> var = Parameter(Tensor(np.array([[1, 2], [3, 4], [5, 6]]), mindspore.float32))
>>> indices = Tensor(np.array([0, 1]), mindspore.int32)
>>> updates = Tensor(np.array([[0.5, 1.0], [1.0, 1.5]]), mindspore.float32)
>>> inplaceIndexAdd = ops.InplaceIndexAdd(axis=0)
>>> var = inplaceIndexAdd(var, indices, updates)
>>> print(var)
[[1.5 3. ]
 [4.  5.5]
 [5.  6. ]]
```

mindspore.ops.InplaceSub

class mindspore.ops.**InplaceSub**(*indices*)

Subtracts v into specified rows of x. Computes $y = x$; $y[i,] - = input\_v$.

Refer to *mindspore.ops.inplace_sub()* for more details.

Parameters

indices (*Union[int, tuple]*) −Indices into the left-most dimension of x, and determines which rows of x to subtract by v. It is an integer or a tuple, whose value is in [0, the first dimension size of x).

Inputs:

- **x** (Tensor) - The tensor to be subtracted. It has shape $(N, *)$ where $*$ means any number of additional dimensions.
- **input_v** (Tensor) - The value tensor subtract from x. It has the same dimension sizes as x except the first dimension, whose size must be the same as *indices*. It has the same data type with x.

Outputs:

Tensor, has the same shape and dtype as x.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> indices = (0, 1)
>>> x = Tensor(np.array([[1, 2], [3, 4], [5, 6]]), mindspore.float32)
>>> input_v = Tensor(np.array([[0.5, 1.0], [1.0, 1.5]]), mindspore.float32)
>>> inplaceSub = ops.InplaceSub(indices)
>>> output = inplaceSub(x, input_v)
>>> print(output)
[[0.5 1. ]
 [2.  2.5]
 [5.  6. ]]
```

mindspore.ops.InplaceUpdate

class mindspore.ops.**InplaceUpdate**(*indices*)

Please use *mindspore.ops.InplaceUpdateV2* instead.

Supported Platforms:

Deprecated

mindspore.ops.InplaceUpdateV2

class mindspore.ops.**InplaceUpdateV2**

Updates specified values in *x* to *v* according to *indices*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.inplace_update()* for more details.

Inputs:

- **x** (Tensor) - A tensor which to be inplace updated. It can be one of the following data types: float32, float16 and int32.
- **indices** (Union[int, tuple]): Indices into the left-most dimension of *x*, and determines which rows of x to update with v. It is an int or tuple, whose value is in [0, the first dimension size of x).
- **v** (Tensor) - A tensor with the same type as *x* and the same dimension size as *x* except the first dimension, which must be the same as the size of *indices*.

Outputs:

Tensor, with the same type and shape as the input *x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> indices = (0, 1)
>>> x = Tensor(np.array([[1, 2], [3, 4], [5, 6]]), mindspore.float32)
>>> v = Tensor(np.array([[0.5, 1.0], [1.0, 1.5]]), mindspore.float32)
>>> inplace_update_v2 = ops.InplaceUpdateV2()
>>> output = inplace_update_v2(x, indices, v)
>>> print(output)
[[0.5 1. ]
 [1.  1.5]
 [5.  6. ]]
```

mindspore.ops.InvertPermutation

class mindspore.ops.**InvertPermutation**

Computes the inverse of an index permutation.

This operator is mainly used to calculate the inverse of index permutation. It requires a 1-dimensional integer tensor x, which represents the index of a zero-based array, and exchanges each value with its index position. In other words, For output tensor y and input tensor x, this operation calculates the following values:

$y[x[i]] = i, \quad i \in [0, 1, \ldots, \text{len}(x) - 1]$.

Note: These values must include 0. There must be no duplicate values and the values can not be negative.

Inputs:

- **input_x** (Union(tuple[int], list[int])) - The input is constructed by multiple integers, i.e., $(y_1, y_2, ..., y_S)$ representing the indices. The values must include 0. There can be no duplicate values or negative values. Only constant value is allowed. The maximum value must be equal to length of input_x.

Outputs:

tuple[int]. It has the same length as the input.

Raises

- **TypeError** –If *input_x* is neither tuple nor list.
- **TypeError** –If element of *input_x* is not an int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> invert = ops.InvertPermutation()
>>> input_data = (3, 4, 0, 2, 1)
>>> output = invert(input_data)
>>> print(output)
(2, 4, 3, 0, 1)
```

mindspore.ops.IsClose

class mindspore.ops.**IsClose**(*rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a tensor of Boolean values indicating whether each element of *input* is "close" to the corresponding element of *other*. Closeness is defined as:

$$|input - other| <= atol + rtol * |other|$$

Refer to *mindspore.ops.isclose()* for more details.

Parameters

- **rtol** (*float, optional*) –Relative tolerance. Default: `1e-05` .

- **atol** (*float, optional*) –Absolute tolerance. Default: `1e-08` .
- **equal_nan** (*bool, optional*) –If `True` , then two NaNs will be considered equal. Default: `False` .

Inputs:

- **input** (Tensor) - First tensor to compare.
- **other** (Tensor) - Second tensor to compare.

Outputs:

Tensor, with the same shape as *input* and *other* after broadcasting, its dtype is bool.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor
>>> from mindspore.ops import IsClose
>>> input = Tensor(np.array([1.3, 2.1, 3.2, 4.1, 5.1]), mindspore.float16)
>>> other = Tensor(np.array([1.3, 3.3, 2.3, 3.1, 5.1]), mindspore.float16)
>>> isclose = IsClose()
>>> output = isclose(input, other)
>>> print(output)
[ True False False False  True]
```

mindspore.ops.Lcm

class mindspore.ops.**Lcm**

Computes least common multiplier of input tensors element-wise. The shape of two inputs should be broadcastable, and data type of them should be one of: int32, int64.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x1** (Tensor) - The first input tensor.
- **x2** (Tensor) - The second input tensor.

Outputs:

Tensor, the shape is the same as the one after broadcasting, and the data type is one with higher digits in the two inputs.

Raises

- **TypeError** –If data type *x1* or *x2* is not int32 or int64.
- **ValueError** –If shape of two inputs are not broadcastable.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x1 = Tensor(np.array([7, 8, 9]))
>>> x2 = Tensor(np.array([14, 6, 12]))
>>> lcm_ = ops.Lcm()
>>> y = lcm_(x1, x2)
>>> print(y)
[14 24 36]
```

mindspore.ops.LeftShift

class mindspore.ops.**LeftShift**

Shift the value of each position of the tensor to the left several bits. The inputs are two tensors, dtypes of them must be consistent, and the shapes of them could be broadcast. The output does not support implicit type conversion.

$$out_i = x_i << y_i$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x1** (Tensor) - The target tensor whose dtype supports all int and uint type, will be shifted to the left by *x2* in element-wise.
- **x2** (Tensor) - The tensor must have the same dtype as *x1*. And the tensor must have the same shape as *x1* or could be broadcast with *x1*.

Outputs:

- **output** (Tensor) - The output tensor, has the same dtype as *x1*. And the shape of the output tensor is the same shape as *x1*, or the same shape as *x1* and *x2* after broadcasting.

Raises

- **TypeError** – If *x1* or *x2* has wrong type.
- **TypeError** – If *x1* or *x2* is not tensor.
- **ValueError** – If *x1* and *x2* could not be broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> left_shift = ops.LeftShift()
>>> x1 = Tensor(np.array([1, 2, 3]).astype(np.int8))
>>> x2 = Tensor(np.array([0, 1, -1]).astype(np.int8))
>>> output = left_shift(x1, x2)
>>> print(output)
[1 4 0]
```

mindspore.ops.LogSpace

class mindspore.ops.**LogSpace**(*steps=10*, *base=10*, *dtype=mstype.float32*)

Generates a 1-D Tensor with a length of steps. The tensor's values are uniformly distributed on a logarithmic scale, ranging from $base^{start}$ to $base^{end}$, including both endpoints. The logarithmic scale is based on the specified *base*.

$$step = (end - start)/(steps - 1)$$
$$output = [base^{start}, base^{start+1*step}, ..., base^{start+(steps-2)*step}, base^{end}]$$

Warning: This is an experimental API that is subject to change or deletion.

Parameters

- **steps** (*int, optional*) – The steps must be a non-negative integer. Default: `10`.
- **base** (*int, optional*) – The base must be a non-negative integer. Default: `10`.
- **dtype** (`mindspore.dtype, optional`) – The dtype of output, include `mstype.float16`, `mstype.float32` or `mstype.float64`. Default: `mstype.float32`.

Inputs:

- **start** (Tensor) - Start value of interval, with shape of 0-D, dtype is float16, float32 or float64.
- **end** (Tensor) - End value of interval, with shape of 0-D, dtype is float16, float32 or float64.

Outputs:

Tensor has the shape as $(step,)$. Its datatype is set by the attr 'dtype'.

Raises

- **TypeError** – If *input* is not a Tensor.
- **TypeError** – If *steps* is not an int.
- **TypeError** – If *base* is not an int.
- **TypeError** – If *dtype* is not mstype.float16, mstype.float32 or mstype.float64.
- **ValueError** – If *steps* is not a non-negative integer.
- **ValueError** – If *base* is not a non-negative integer.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> logspace = ops.LogSpace(steps = 10, base = 10, dtype=mstype.float32)
>>> start = Tensor(1, mstype.float32)
>>> end = Tensor(10, mstype.float32)
>>> output = logspace(start, end)
>>> print(output)
[1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.e+10]
```

mindspore.ops.LuUnpack

class mindspore.ops.**LuUnpack**(*unpack_data=True*, *unpack_pivots=True*)

Converts *LU_data* and *LU_pivots* back into P, L and U matrices, where P is a permutation matrix, L is a lower triangular matrix, and U is an upper triangular matrix. Typically, *LU_data* and *LU_pivots* are generated from the LU decomposition of a matrix.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.lu_unpack()* for more details.

Parameters

- **unpack_data** (*bool, optional*) – A flag indicating if the LU_data should be unpacked. If `False` , then the returned L and U are None. Default: `True` .
- **unpack_pivots** (*bool, optional*) – A flag indicating if the LU_pivots should be unpacked into a permutation matrix P. If `False` , then the returned P is None. Default: `True` .

Inputs:

- **LU_data** (Tensor) - The packed LU factorization data. The shape of a tensor is $(*, M, N)$, where $*$ is batch dimensions, with data type int8, uint8, int16, int32, int64, float16, float32, float64. The dims of LU_data must be equal to or greater than 2.
- **LU_pivots** (Tensor) - The packed LU factorization pivots. The shape of a tensor is $(*, min(M, N))$, where $*$ is batch dimensions, with data type int8, uint8, int16, int32, int64.

Outputs:

- **pivots** (Tensor) - The permutation matrix of LU factorization. The shape is $(*, M, M)$, the dtype is same as *LU_data*.
- **L** (Tensor) - The L matrix of LU factorization. The dtype is the same as *LU_data*.
- **U** (Tensor) - The U matrix of LU factorization. The dtype is the same as *LU_data*.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> LU_data = Tensor(np.array([[[-0.3806, -0.4872,  0.5536],
...                             [-0.1287,  0.6508, -0.2396],
...                             [ 0.2583,  0.5239,  0.6902]],
...                             [[ 0.6706, -1.1782,  0.4574],
...                             [-0.6401, -0.4779,  0.6701],
...                             [ 0.1015, -0.5363,  0.6165]]]), mstype.float32)
>>> LU_pivots = Tensor(np.array([[1, 3, 3],
...                              [2, 3, 3]]), mstype.int32)
>>> lu_unpack = ops.LuUnpack()
>>> pivots, L, U = lu_unpack(LU_data, LU_pivots)
>>> print(pivots)
[[[1. 0. 0.]
  [0. 0. 1.]
  [0. 1. 0.]]

 [[0. 0. 1.]
  [1. 0. 0.]
  [0. 1. 0.]]]
>>> print(L)
[[[ 1.      0.      0.    ]
  [-0.1287  1.      0.    ]
  [ 0.2583  0.5239  1.    ]]

 [[ 1.      0.      0.    ]
  [-0.6401  1.      0.    ]
  [ 0.1015 -0.5363  1.    ]]]
>>> print(U)
[[[-0.3806 -0.4872  0.5536]
  [ 0.      0.6508 -0.2396]
  [ 0.      0.      0.6902]]

 [[ 0.6706 -1.1782  0.4574]
  [ 0.     -0.4779  0.6701]
  [ 0.      0.      0.6165]]]
```

mindspore.ops.MaskedFill

class mindspore.ops.**MaskedFill**

```
prim = ops.MaskedFill()
out = prim(input_x, mask, value)
```

is equivalent to

```
ops.masked_fill(input_x, mask, value)
```

Refer to *mindspore.ops.masked_fill()* for more details.

mindspore.ops.MaskedScatter

class mindspore.ops.**MaskedScatter**

Updates the value in the input with value in *updates* according to the *mask*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor): The input Tensor to be updated.
- **mask** (Tensor[bool]): The mask Tensor indicating which elements should be modified or replaced. The shapes of *mask* and *x* must be the same or broadcastable.
- **updates** (Tensor): The values to scatter into the target tensor *x*. It has the same data type as *x*. The number of elements must be greater than or equal to the number of True's in *mask*.

Outputs:

Tensor, with the same type and shape as *x*.

Raises

- **TypeError** –If *x*, *mask* or *updates* is not a Tensor.
- **TypeError** –If data type of *x* is not be supported.
- **TypeError** –If dtype of *mask* is not bool.
- **TypeError** –If the dim of *x* less than the dim of *mask*.
- **ValueError** –If *mask* can not be broadcastable to *x*.
- **ValueError** –If the number of elements in *updates* is less than number of True's in *mask*.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([1., 2., 3., 4.]), mindspore.float32)
>>> mask = Tensor(np.array([True, True, False, True]), mindspore.bool_)
>>> updates = Tensor(np.array([5., 6., 7.]), mindspore.float32)
>>> output = ops.MaskedScatter()(input_x, mask, updates)
>>> print(output)
[5. 6. 3. 7.]
```

mindspore.ops.MaskedSelect

class mindspore.ops.**MaskedSelect**

```
prim = ops.MaskedSelect()
out = prim(input, mask)
```

is equivalent to

```
ops.masked_select(input, mask)
```

Refer to *mindspore.ops.masked_select()* for more details.

mindspore.ops.MatrixBandPart

class mindspore.ops.**MatrixBandPart**

Extracts the central diagonal band of each matrix in a tensor, with all values outside the central band set to zero.

Refer to *mindspore.ops.matrix_band_part()* for more details.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **x** (Tensor) - Input tensor. $(*, m, n)$ where $*$ means, any number of additional dimensions.
- **lower** (Union[int, Tensor]) - Number of subdiagonals to keep. The data type must be int32 or int64. If negative, keep entire lower triangle.
- **upper** (Union[int, Tensor]) - Number of superdiagonals to keep. The data type must be int32 or int64. If negative, keep entire upper triangle.

Outputs:

Tensor, has the same type and shape as x.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> matrix_band_part = ops.MatrixBandPart()
>>> x = np.ones([2, 4, 4]).astype(np.float32)
>>> output = matrix_band_part(Tensor(x), 2, 1)
>>> print(output)
[[[1. 1. 0. 0.]
  [1. 1. 1. 0.]
  [1. 1. 1. 1.]
  [0. 1. 1. 1.]]
 [[1. 1. 0. 0.]
  [1. 1. 1. 0.]
  [1. 1. 1. 1.]
  [0. 1. 1. 1.]]]
```

mindspore.ops.MatrixDiagPartV3

class mindspore.ops.**MatrixDiagPartV3**(*align='RIGHT_LEFT'*)

Returns the diagonal part of a tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.matrix_diag_part()* for more details.

Parameters

align (*str, optional*) – specifies how superdiagonals and subdiagonals should be aligned. Supported values: `"RIGHT_LEFT"` , `"LEFT_RIGHT"` , `"LEFT_LEFT"` , `"RIGHT_RIGHT"` . Default: `"RIGHT_LEFT"` .

- When set to `"RIGHT_LEFT"` , the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
- When set to `"LEFT_RIGHT"` , the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
- When set to `"LEFT_LEFT"` , the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
- When set to `"RIGHT_RIGHT"` , the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Inputs:

- **x** (Tensor) - Rank r, where r >= 2.
- **k** (Tensor) - A Tensor of type int32. Diagonal offset(s). Positive value means superdiagonal, 0 refers to the main diagonal, and negative value means subdiagonals. k can be a single integer (for a single diagonal) or a pair of integers specifying the low and high ends of a matrix band. k[0] must not be larger than k[1]. The value of k has restructions, meaning value of k must be in (-x.shape[-2], x.shape[-1]).
- **padding_value** (Tensor) - A Tensor. Have the same dtype as x. The number to fill the area outside the specified diagonal band with. There must be only one value.

Outputs:

A Tensor. Has the same type as *x*. Assume *x* has r dimensions $(I, J, ..., M, N)$. Let *max_diag_len* be the maximum length among all diagonals to be extracted, $max\_diag\_len = min(M + min(k[1], 0), N + min(-k[0], 0))$ Let *num_diags* be the number of diagonals to extract, $num\_diags = k[1] - k[0] + 1$. If $num\_diags == 1$, the output tensor is of rank r - 1 with shape $(I, J, ..., L, max\_diag\_len)$ Otherwise, the output tensor has rank r with dimensions $(I, J, ..., L, num\_diags, max\_diag\_len)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1, 2, 3, 4],
...                      [5, 6, 7, 8],
...                      [9, 8, 7, 6]]), mindspore.float32)
>>> k =Tensor(np.array([1, 3]), mindspore.int32)
>>> padding_value = Tensor(np.array(9), mindspore.float32)
>>> matrix_diag_part_v3 = ops.MatrixDiagPartV3(align='RIGHT_LEFT')
>>> output = matrix_diag_part_v3(x, k, padding_value)
>>> print(output)
[[9. 9. 4.]
 [9. 3. 8.]
 [2. 7. 6.]]
>>> print(output.shape)
(3, 3)
```

mindspore.ops.MatrixDiagV3

class mindspore.ops.**MatrixDiagV3**(*align='RIGHT_LEFT'*)

Constructs a diagonal matrix or a batch of diagonal matrices from a given input Tensor.

Warning: This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.matrix_diag()* for more details.

Parameters

align (*str, optional*) – specifies how superdiagonals and subdiagonals should be aligned. Supported values: "RIGHT_LEFT", "LEFT_RIGHT", "LEFT_LEFT", "RIGHT_RIGHT". Default: "RIGHT_LEFT".

- When set to "RIGHT_LEFT", the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
- When set to "LEFT_RIGHT", the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
- When set to "LEFT_LEFT", the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
- When set to "RIGHT_RIGHT", the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Inputs:

- **x** (Tensor) - The diagonal Tensor.
- **k** (Union[int, Tensor], optional) - Diagonal offsets. A Tensor of type int32. Positive value means superdiagonal, 0 refers to the main diagonal, and negative value means subdiagonals. *k* can be a single integer (for a single diagonal) or a pair of integers specifying the low and high ends of a matrix band. k[0] must not be larger than k[1]. The value must be in the range of given or derivated *num_rows* and *num_cols*, meaning value of k must be in (-num_rows, num_cols). Default: 0.
- **num_rows** (Union[int, Tensor], optional) - The number of rows of the output Tensor. A Tensor of type int32 with only one value. If *num_rows* is -1, indicating that the innermost matrix of the output Tensor is a square matrix, and the

real number of rows will be derivated by other inputs. That is $num\_rows = x.shape[-1] - min(k[1], 0)$. Otherwise, the value must be equal or greater than $x.shape[-1] - min(k[1], 0)$. Default: -1.

- **num_cols** (Union[int, Tensor], optional) - The number of columns of the output Tensor. A Tensor of type int32 with only one value. If *num_cols* is -1, indicating that the innermost matrix of the output Tensor is a square matrix, and the real number of columns will be derivated by other inputs. That is $num\_cols = x.shape[-1] + max(k[0], 0)$. Otherwise, the value must be equal or greater than $x.shape[-1] - min(k[1], 0)$. Default: -1.
- **padding_value** (Union[int, float, Tensor], optional) - The number to fill the area outside the specified diagonal band. A Tensor with only one value. Have the same dtype as x. Default: `0` .

Outputs:

A Tensor. Has the same type as *x*. Suppose *x* has r dimensions with shape $(I, J, ..., M, N)$. The output Tensor has rank r + 1 with shape $(I, J, ..., M, num\_rows, num\_cols)$ when only one diagonal is given (k is an integer or k[0] == k[1]). Otherwise, it has rank r with shape $(I, J, ..., num\_rows, num\_cols)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[8, 9, 0],
...                      [1, 2, 3],
...                      [0, 4, 5]]), mindspore.float32)
>>> k =Tensor(np.array([-1, 1]), mindspore.int32)
>>> num_rows = Tensor(np.array(3), mindspore.int32)
>>> num_cols = Tensor(np.array(3), mindspore.int32)
>>> padding_value = Tensor(np.array(11), mindspore.float32)
>>> matrix_diag_v3 = ops.MatrixDiagV3(align='LEFT_RIGHT')
>>> output = matrix_diag_v3(x, k, num_rows, num_cols, padding_value)
>>> print(output)
[[ 1.  8. 11.]
 [ 4.  2.  9.]
 [11.  5.  3.]]
>>> print(output.shape)
(3, 3)
```

mindspore.ops.MatrixSetDiagV3

class mindspore.ops.**MatrixSetDiagV3**(*align='RIGHT_LEFT'*)

Updates the diagonal part of a batched tensor. It takes a Tensor *x* and *diagonal* as input and returns a Tensor in which the specified diagonal values in the innermost matrices will be replaced by the values in the *diagonal*.

Diagonals shorter than *max_diag_len* need to be padded, where *max_diag_len* is the longest diagonal value. The dimension of *diagonal* is $shape[-2]$ must be equal to num_diags calculated by $num\_diags = k[1] - k[0] + 1$. The dimension of *diagonal* is $shape[-1]$ must be equal to the longest diagonal value *max_diag_len* calculated by $max\_diag\_len = min(x.shape[-2] + min(k[1], 0), x.shape[-1] + min(-k[0], 0))$.

Assume x is an n-D Tensor with shape $(d_1, d_2, ..., d_{n-2}, d_{n-1}, d_n)$. If k is an integer or $k[0] == k[1]$, *diagonal* is an (n-1)-D Tensor with shape $(d_1, d_2, ..., d_{n-2}, max\_diag\_len)$ Otherwise, it has the same rank as x with shape $(d_1, d_2, ..., d_{n-2}, num\_diags, max\_diag\_len)$.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

align (*str, optional*) –specifies how superdiagonals and subdiagonals should be aligned. Supported values: "RIGHT_LEFT" , "LEFT_RIGHT", "LEFT_LEFT" , "RIGHT_RIGHT" . Default: "RIGHT_LEFT" .

- When set to "RIGHT_LEFT" , the alignment of superdiagonals will be towards the right side (padding the row on the left), while subdiagonals will be towards the left side (padding the row on the right)
- When set to "LEFT_RIGHT" , the alignment of superdiagonals will be towards the left side (padding the row on the right), while subdiagonals will be towards the right side (padding the row on the left)
- When set to "LEFT_LEFT" , the alignment of both superdiagonals and subdiagonals will be towards the left side(padding the row on the right).
- When set to "RIGHT_RIGHT" , the alignment of both superdiagonals and subdiagonals will be towards the right side(padding the row on the left).

Inputs:

- **x** (Tensor) - A n-D Tensor, where $n >= 2$.
- **diagonal** (Tensor) - A Tensor with the same dtype as *x*. Its rank depends on *k*. If *k* is an integer or $k[0] == k[1]$, its dimension is $n - 1$. Otherwise, it has dimension n.
- **k** (Tensor) - Diagonal offset(s), Tensor of type int32. *k* can either be a single integer, which represents a single diagonal, or a pair of integers that specify the low and high ends of a matrix band. In this case, *k[0]* should not be greater than *k[1]*. The value of *k* has restructions, which means that value of *k* must be in range $(-x.shape[-2], x.shape[-1])$. Input *k* must be const Tensor when taking Graph mode.
  - *k* > *0* refers to a superdiagonal.
  - *k* = *0* refers to the main diagonal.
  - *k* < *0* refers to subdiagonals.

Outputs:

Tensor. The same type and shape as *x*.

Raises

- **TypeError** –If any input is not Tensor.
- **TypeError** –If input *x* and *diagonal* are not the same dtype.
- **TypeError** –If *k* is not int32 dtype.
- **ValueError** –If *align* is not a string or not in the valid range.
- **ValueError** –If rank of *k* is not equal to 0 or 1.
- **ValueError** –If rank of *x* is not greater equal to 2.
- **ValueError** –If size of *k* is not equal to 1 or 2.
- **ValueError** –If *k[1]* is not greater equal to *k[0]* in case the size of *k* is 2.
- **ValueError** –If the *diagonal* rank size don't match with input *x* rank size.
- **ValueError** –If the *diagonal* shape value don't match with input *x* shape value.
- **ValueError** –If the diagonal $shape[-2]$ is not equal to num_diags calculated by $num\_diags = k[1] - k[0] + 1$.

- **ValueError** –If the value of k is not in $(-x.shape[-2], x.shape[-1])$.
- **ValueError** –If the diagonal $shape[-1]$ is not equal to the max_diag_len calculated by $max\_diag\_len = min(x.shape[-2] + min(k[1], 0), x.shape[-1] + min(-k[0], 0))$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[7, 7, 7, 7],
...                      [7, 7, 7, 7],
...                      [7, 7, 7, 7]]), mindspore.float32)
>>> diagonal = Tensor(np.array([[0, 9, 1],
...                             [6, 5, 8],
...                             [1, 2, 3],
...                             [4, 5, 0]]), mindspore.float32)
>>> k =Tensor(np.array([-1, 2]), mindspore.int32)
>>> matrix_set_diag_v3 = ops.MatrixSetDiagV3(align='RIGHT_LEFT')
>>> output = matrix_set_diag_v3(x, diagonal, k)
>>> print(output)
[[1. 6. 9. 7.]
 [4. 2. 5. 1.]
 [7. 5. 3. 8.]]
>>> print(output.shape)
(3, 4)
```

mindspore.ops.MatrixSolve

class mindspore.ops.**MatrixSolve**(*adjoint=False*)

Solves systems of linear equations.

Parameters

adjoint (*bool, optional*) –Indicates whether the adjoint of the matrix is used during the computation. Default: `False` , use its transpose instead.

Inputs:

- **matrix** (Tensor) - A tensor of shape $(..., M, M)$, is a matrix of coefficients for a system of linear equations.
- **rhs** (Tensor) - A tensor of shape $(..., M, K)$, is a matrix of the resulting values of a system of linear equations. *rhs* must have the same type as *matrix*.

Outputs:

Tensor, a matrix composed of solutions to a system of linear equations, which has the same type and shape as *rhs*.

Raises

- **TypeError** –If *adjoint* is not the type of bool.
- **TypeError** –If the type of *matrix* is not one of the following dtype: mstype.float16, mstype.float32, mstype.float64, mstype.complex64, mstype.complex128.

- **TypeError** –If the type of *matrix* is not the same as that of *rhs*.
- **ValueError** –If the rank of *matrix* less than 2.
- **ValueError** –If the dimension of *matrix* is not the same as *rhs* .
- **ValueError** –If the inner-most 2 dimension of *matrix* is not the same.
- **ValueError** –If the inner-most 2 dimension of *rhs* does not match *matrix* .

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> matrix = Tensor(np.array([[1.0  , 4.0],
...                           [2.0 , 7.0]]), mindspore.float32)
>>> rhs = Tensor(np.array([[1.0]  , [3.0]]), mindspore.float32)
>>> matrix_solve = ops.MatrixSolve(adjoint = False)
>>> output = matrix_solve(matrix, rhs)
>>> print(output)
[[5.0]
 [-1.0]]
```

mindspore.ops.Meshgrid

class mindspore.ops.**Meshgrid**(*indexing='xy'*)

Generates coordinate matrices from given coordinate tensors.

Refer to *mindspore.ops.meshgrid()* for more details.

Parameters

indexing (*str, optional*) –Cartesian `'xy'` or matrix ('ij') indexing of output. Valid options: xy' or `'ij'`. In the 2-D case with inputs of length M and N, for `'xy'` indexing, the shape of outputs is (N, M) for `'ij'` indexing, the shape of outputs is (M, N). In the 3-D case with inputs of length M, N and P, for `'xy'` indexing, the shape of outputs is (N, M, P) and for `'ij'` indexing, the shape of outputs is (M, N, P). Default: `'xy'` .

Inputs:

- **inputs** (Union(tuple[Tensor], list[Tensor])) - In GRAPH_MODE, a tuple of N 1-D Tensor objects and the length of input should be greater than 1. In PYNATIVE_MODE, a tuple of N 0-D or 1-D Tensor objects and the length of input should be greater than 0. The data type is Number.

Outputs:

Tensors, A Tuple of N N-D Tensor objects. The data type is the same with the Inputs.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([1, 2, 3, 4]).astype(np.int32))
>>> y = Tensor(np.array([5, 6, 7]).astype(np.int32))
>>> z = Tensor(np.array([8, 9, 0, 1, 2]).astype(np.int32))
>>> inputs = (x, y, z)
>>> meshgrid = ops.Meshgrid(indexing='xy')
>>> output = meshgrid(inputs)
>>> print(output)
(Tensor(shape=[3, 4, 5], dtype=Int32, value=
[[[1, 1, 1, 1, 1],
  [2, 2, 2, 2, 2],
  [3, 3, 3, 3, 3],
  [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
  [2, 2, 2, 2, 2],
  [3, 3, 3, 3, 3],
  [4, 4, 4, 4, 4]],
  [[1, 1, 1, 1, 1],
  [2, 2, 2, 2, 2],
  [3, 3, 3, 3, 3],
  [4, 4, 4, 4, 4]]]),
Tensor(shape=[3, 4, 5], dtype=Int32, value=
[[[5, 5, 5, 5, 5],
  [5, 5, 5, 5, 5],
  [5, 5, 5, 5, 5],
  [5, 5, 5, 5, 5]],
  [[6, 6, 6, 6, 6],
  [6, 6, 6, 6, 6],
  [6, 6, 6, 6, 6],
  [6, 6, 6, 6, 6]],
  [[7, 7, 7, 7, 7],
  [7, 7, 7, 7, 7],
  [7, 7, 7, 7, 7],
  [7, 7, 7, 7, 7]]]),
Tensor(shape=[3, 4, 5], dtype=Int32, value=
[[[8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2]],
  [[8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2],
  [8, 9, 0, 1, 2]]]))
```

mindspore.ops.Mvlgamma

class mindspore.ops.**Mvlgamma**(*p*)

Calculates the multivariate log-gamma function element-wise for a given dimension *p*.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.mvlgamma()* for more details.

Parameters

p (*int*) – The number of dimensions. And the value of *p* must be greater than or equal to 1.

Inputs:

- **x** (Tensor) - The tensor to compute the multivariate log-gamma function, which must be one of the following types: float32, float64. The shape is $(N, *)$, where $*$ means any number of additional dimensions. And the value of any element in *x* must be greater than $(p - 1)/2$.

Outputs:

Tensor, has the same shape and type as *x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[3, 4, 5], [4, 2, 6]]), mindspore.float32)
>>> op = ops.Mvlgamma(p=3)
>>> y = op(x)
>>> print(y)
[[ 2.694925   5.402975   9.140645 ]
 [ 5.402975   1.5963125 13.640454 ]]
```

mindspore.ops.NanToNum

class mindspore.ops.**NanToNum**(*nan=None*, *posinf=None*, *neginf=None*)

```
prim = ops.NanToNum(nan, posinf, neginf)
out = prim(input)
```

is equivalent to

```
ops.nan_to_num(input, nan, posinf, neginf)
```

Refer to *mindspore.ops.nan_to_num()* for more details.

mindspore.ops.NonZero

class mindspore.ops.**NonZero**

Return a Tensor of the positions of all non-zero values.

Inputs:

- **input** (Tensor) - The input Tensor.
  - Ascend: its rank can be equal to 0 except O2 mode.
  - CPU/GPU: its rank should be greater than or eaqual to 1.

Outputs:

Tensor, a 2-D Tensor whose data type is int64, containing the positions of all non-zero values of the input. If the dimension of *input* is *D* and the number of non-zero in *input* is *N* , then the shape of output is (N, D) .

Raises

- **TypeError** –If *input* is not Tensor.
- **RuntimeError** –On GPU or CPU or Ascend O2 mode, if dim of *input* is equal to 0.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> input = mindspore.tensor([1, 0, 2, 0, 3], mindspore.int32)
>>> output = mindspore.ops.NonZero()(input)
>>> print(output)
[[0]
 [2]
 [4]]
```

mindspore.ops.ParallelConcat

class mindspore.ops.**ParallelConcat**

Concats input tensors along the first dimension.

The difference between Concat and ParallelConcat is that Concat requires all of the inputs be computed before the operation will begin but doesn't require that the input shapes be known during graph construction. Parallel concat will copy pieces of the input into the output as they become available, in some situations this can provide a performance benefit.

Note: The input tensors are all required to have size 1 in the first dimension.

Inputs:

- **values** (tuple, list) - A tuple or a list of input tensors. The data type and shape of these tensors must be the same and their rank should not be less than 1. The supported date type is Number on CPU, the same for Ascend except [float64, complex64, complex128].

Outputs:

Tensor, data type is the same as *values*.

Raises

- **TypeError** –If any type of the inputs is not a Tensor.
- **TypeError** –If the data type of these tensors are not the same.
- **ValueError** –If any tensor.shape[0] is not 1.
- **ValueError** –If rank of any Tensor in *values* is less than 1.
- **ValueError** –If the shape of these tensors are not the same.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> data1 = Tensor(np.array([[0, 1]]).astype(np.int32))
>>> data2 = Tensor(np.array([[2, 1]]).astype(np.int32))
>>> op = ops.ParallelConcat()
>>> output = op((data1, data2))
>>> print(output)
[[0 1]
 [2 1]]
```

mindspore.ops.PopulationCount

class mindspore.ops.**PopulationCount**

Computes element-wise population count(a.k.a bitsum, bitcount).

Refer to *mindspore.ops.population_count()* for more details.

Inputs:

- **input_x** (Tensor) - Tensor of any dimension. The data type must be int16 or uint16 (Ascend). The data type must be int8, int16, int32, int64, uint8, uint16, uint32, uint64 (CPU and GPU).

Outputs:

Tensor, with the same shape as the input, and the data type is uint8.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> input_x = Tensor([0, 1, 3], mindspore.int16)
>>> output = ops.PopulationCount()(input_x)
>>> print(output)
[0 1 2]
```

mindspore.ops.RandomShuffle

class mindspore.ops.**RandomShuffle**(*seed=0*, *seed2=0*)

Randomly shuffles a Tensor along its first dimension.

Note:

- Random seed: a set of regular random numbers can be obtained through some complex mathematical algorithms, and the random seed determines the initial value of this random number. If the random seed is the same in two separate calls, the random number generated will not change.
- Using the Philox algorithm to scramble seed and seed2 to obtain random seed so that the user doesn't need to worry about which seed is more important.

Parameters

- **seed** (*int, optional*) – The operator-level random seed, used to generate random numbers, must be non-negative. Default: `0` .
- **seed2** (*int, optional*) – The global random seed, which combines with the operator-level random seed to determine the final generated random number, must be non-negative. Default: `0` .

Inputs:

- **x** (Tensor) - The Tensor need be shuffled.

Outputs:

Tensor. The shape and type are the same as the input *x*.

Raises

TypeError – If data type of *seed* or *seed2* is not int.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 2, 3, 4]), mstype.float32)
>>> shuffle = ops.RandomShuffle(seed=1, seed2=1)
>>> output = shuffle(x)
>>> print(output.shape)
(4,)
```

mindspore.ops.Range

class mindspore.ops.**Range**(*maxlen=1000000*)

```
prim = ops.Range(maxlen)
out = prim(start, end, step)
```

is equivalent to

```
ops.range(start, end, step, maxlen)
```

Refer to *mindspore.ops.range()* for more details.

mindspore.ops.Rank

class mindspore.ops.**Rank**

Returns the rank of a tensor.

Refer to *mindspore.ops.rank()* for more details.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_tensor = Tensor(np.array([[2, 2], [2, 2]]), mindspore.float32)
>>> rank = ops.Rank()
>>> output = rank(input_tensor)
>>> print(output)
2
>>> print(type(output))
<class 'int'>
```

mindspore.ops.Renorm

class mindspore.ops.**Renorm**(*p*, *dim*, *maxnorm*)

Renormalizes the sub-tensors along dimension *dim*, and each sub-tensor's p-norm should not exceed the 'maxnorm'. The values of current sub-tensor don't need change if the p-norm of the sub-tensor is less than *maxnorm*. Otherwise the sub-tensor needs to be modified to the original value of the corresponding position divided by the p-norm of the substensor and then multiplied by *maxnorm*.

Refer to *mindspore.ops.renorm()* for more details.

Parameters

- **p** (*int*) –Power of norm calculation.
- **dim** (*int*) –The dimension that expected to get the slice-tensor.
- **maxnorm** (*float32*) –Max norm.

Inputs:

- **x** (Tensor) - A Tensor, types: float32 or float16.

Outputs:

Tensor, has the same dtype and shape as input.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]]), mindspore.float32)
>>> y = ops.Renorm(p=1, dim=0, maxnorm=5.)(x)
>>> print(y)
[[1.       1.        1.        ]
[1.6666666 1.6666666 1.6666666 ]
[1.6666667 1.6666667 1.6666667 ]]
```

mindspore.ops.Reshape

class mindspore.ops.**Reshape**

```
prim = ops.Reshape()
out = prim(input, shape)
```

is equivalent to

```
ops.reshape(input, shape)
```

Refer to *mindspore.ops.reshape()* for more details.

mindspore.ops.ReverseSequence

class mindspore.ops.**ReverseSequence**(*seq_dim*, *batch_dim=0*)

Reverses variable length slices.

Parameters

- **seq_dim** (*int*) –The dimension where reversal is performed. Required.
- **batch_dim** (*int, optional*) –The input is sliced in this dimension. Default: `0` .

Inputs:

- **x** (Tensor) - The input to reverse, supporting all number types including bool.
- **seq_lengths** (Tensor) - Must be a 1-D vector with int32 or int64 types.

Outputs:

Tensor, with the same shape and data type as *x*.

Raises

- **TypeError** –If *seq_dim* or *batch_dim* is not an int.
- **ValueError** –If $len(seq\_lengths) != x.shape[batch\_dim]$.
- **ValueError** –If $batch\_dim == seq\_dim$.
- **ValueError** –If $seq\_dim < 0$ or $seq\_dim >= len(x.shape)$.
- **ValueError** –If $batch\_dim < 0$ or $batch\_dim >= len(x.shape)$.
- **RuntimeError** –If any value of *seq_lengths* is less than 0.
- **RuntimeError** –If any value of *seq_lengths* is larger than *x.shape[seq_dim]*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> seq_lengths = Tensor(np.array([1, 2, 3]))
>>> reverse_sequence = ops.ReverseSequence(seq_dim=1)
>>> output = reverse_sequence(x, seq_lengths)
>>> print(output)
[[1. 2. 3.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> seq_lengths = Tensor(np.array([1, 2, 3]))
>>> reverse_sequence = ops.ReverseSequence(seq_dim=0, batch_dim=1)
>>> output = reverse_sequence(x, seq_lengths)
>>> print(output)
[[1. 5. 9.]
 [4. 2. 6.]
```

(continues on next page)

(continued from previous page)

```
 [7. 8. 3.]]
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> seq_lengths = Tensor(np.array([2, 2, 3]))
>>> reverse_sequence = ops.ReverseSequence(seq_dim=1)
>>> output = reverse_sequence(x, seq_lengths)
>>> print(output)
[[2. 1. 3.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), mindspore.float32)
>>> seq_lengths = Tensor(np.array([3, 2, 3]))
>>> reverse_sequence = ops.ReverseSequence(seq_dim=1)
>>> output = reverse_sequence(x, seq_lengths)
>>> print(output)
[[3. 2. 1.]
 [5. 4. 6.]
 [9. 8. 7.]]
>>> x = Tensor(np.array([[1, 2, 3, 4], [5, 6, 7, 8]]), mindspore.float32)
>>> seq_lengths = Tensor(np.array([4, 4]))
>>> reverse_sequence = ops.ReverseSequence(seq_dim=1)
>>> output = reverse_sequence(x, seq_lengths)
>>> print(output)
[[4. 3. 2. 1.]
 [8. 7. 6. 5.]]
```

mindspore.ops.ReverseV2

class mindspore.ops.**ReverseV2**(*axis*)

```
prim = ops.ReverseV2(axis)
out = prim(input)
```

is equivalent to

```
ops.flip(input, axis)
```

Refer to *mindspore.ops.flip()* for more details.

mindspore.ops.RightShift

class mindspore.ops.**RightShift**

Shift the value of each position of Tensor *input_x* to the right by corresponding bits in Tensor *input_y*. The inputs are two tensors, dtypes of them must be consistent, and the shapes of them could be broadcast.

$$out_i = x_i >> y_i$$

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **input_x** (Tensor) - The target tensor, will be shifted to the right by *input_y* bits element-wise. Support all int and uint types.

- **input_y** (Tensor) - Number of bits shifted, the tensor must have the same type as *input_x*.

Outputs:

- **output** (Tensor) - The output tensor, has the same type as *input_x*.

Raises

- **TypeError** –If *input_x* or *input_y* is not tensor.
- **TypeError** –If *input_x* and *input_y* could not be broadcast.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([1, 2, 3]).astype(np.uint8))
>>> input_y = Tensor(np.array([1, 1, 1]).astype(np.uint8))
>>> output = ops.RightShift()(input_x, input_y)
>>> print(output)
[0 1 1]
```

mindspore.ops.ScatterNd

class mindspore.ops.**ScatterNd**

```
prim = ops.ScatterNd()
out = prim(indices, updates, shape)
```

is equivalent to

```
ops.scatter_nd(indices, updates, shape)
```

Refer to *mindspore.ops.scatter_nd()* for more details.

mindspore.ops.ScatterNdDiv

class mindspore.ops.**ScatterNdDiv**(*use_locking=False*)

Applies sparse division to individual values or slices in a tensor.

Using given values to update tensor value through the division operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.scatter_nd_div()* for more details.

Parameters

use_locking (*bool, optional*) –Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor.
- **indices** (Tensor) - The index to do div operation whose data type must be int32 or int64. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor to do the div operation with *input_x*. The data type is same as *input_x*, and the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8]), mindspore.float32), name=
 ↪"x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> use_locking = False
>>> scatter_nd_div = ops.ScatterNdDiv(use_locking)
>>> output = scatter_nd_div(input_x, indices, updates)
>>> print(output)
[1.         0.25       0.5        4.         0.71428573 6.
 7.         0.8888889 ]
>>> input_x = Parameter(Tensor(np.ones((4, 4, 4)), mindspore.float32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
 ↪mindspore.float32)
>>> use_locking = False
>>> scatter_nd_div = ops.ScatterNdDiv(use_locking)
>>> output = scatter_nd_div(input_x, indices, updates)
>>> print(output)
[[[1.         1.         1.         1.        ]
  [0.5        0.5        0.5        0.5       ]
  [0.33333334 0.33333334 0.33333334 0.33333334]
  [0.25       0.25       0.25       0.25      ]]
 [[1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]]
 [[0.2        0.2        0.2        0.2       ]
  [0.16666667 0.16666667 0.16666667 0.16666667]
  [0.14285715 0.14285715 0.14285715 0.14285715]
  [0.125      0.125      0.125      0.125     ]]
 [[1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]
  [1.         1.         1.         1.        ]]]
```

mindspore.ops.ScatterNdMax

class mindspore.ops.**ScatterNdMax**(*use_locking=False*)

Computes sparse maximum to individual values or slices in a tensor.

Using given values to update parameter value through the maximum operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

Refer to *mindspore.ops.scatter_nd_max()* for more details.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor.
- **indices** (Tensor) - The index to do maximum operation whose data type must be int32 or int64. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor to do the max operation with *input_x*. The data type is same as *input_x*, and the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8]), mindspore.float32), name=
 ↪"x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> scatter_nd_max = ops.ScatterNdMax()
>>> output = scatter_nd_max(input_x, indices, updates)
>>> print(output)
[ 1. 8. 6.  4. 7.  6.  7. 9.]
>>> input_x = Parameter(Tensor(np.ones((4, 4, 4)), mindspore.int32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
 ↪mindspore.int32)
>>> scatter_nd_max = ops.ScatterNdMax()
>>> output = scatter_nd_max(input_x, indices, updates)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
```

(continues on next page)

(continued from previous page)

```
  [1 1 1 1]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
```

mindspore.ops.ScatterNdMin

class mindspore.ops.**ScatterNdMin**(*use_locking=False*)

Applies sparse minimum to individual values or slices in a tensor.

Using given values to update tensor value through the minimum operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

Refer to *mindspore.ops.scatter_nd_min()* for more details.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor.
- **indices** (Tensor) - The index to do minimum operation whose data type must be int32 or int64. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor to do the max operation with *input_x*. The data type is same as *input_x*, and the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.ones(8) * 10, mindspore.float32), name="x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> use_locking = False
>>> scatter_nd_min = ops.ScatterNdMin(use_locking)
>>> output = scatter_nd_min(input_x, indices, updates)
>>> print(output)
[10.  8.  6. 10.  7. 10. 10.  9.]
>>> input_x = Parameter(Tensor(np.ones((4, 4, 4)) * 10, mindspore.int32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
```

(continues on next page)

(continued from previous page)

```
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
↪mindspore.int32)
>>> use_locking = False
>>> scatter_nd_min = ops.ScatterNdMin(use_locking)
>>> output = scatter_nd_min(input_x, indices, updates)
>>> print(output)
[[[ 1  1  1  1]
  [ 2  2  2  2]
  [ 3  3  3  3]
  [ 4  4  4  4]]
 [[10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]]
 [[ 5  5  5  5]
  [ 6  6  6  6]
  [ 7  7  7  7]
  [ 8  8  8  8]]
 [[10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]
  [10 10 10 10]]]
```

mindspore.ops.ScatterNdMul

class mindspore.ops.**ScatterNdMul**(*use_locking=False*)

Applies sparse multiplication to individual values or slices in a tensor.

Using given values to update parameter value through the multiplication operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.scatter_nd_mul()* for more details.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor.
- **indices** (Tensor) - The index to do mul operation whose data type must be int32 or int64. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor to do the mul operation with *input_x*. The data type is same as *input_x*, and the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8]), mindspore.float32), name=
 ↪"x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> scatter_nd_mul = ops.ScatterNdMul()
>>> output = scatter_nd_mul(input_x, indices, updates)
>>> print(output)
[ 1. 16. 18.  4. 35.  6.  7. 72.]
>>> input_x = Parameter(Tensor(np.ones((4, 4, 4)), mindspore.int32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
 ↪mindspore.int32)
>>> scatter_nd_mul = ops.ScatterNdMul()
>>> output = scatter_nd_mul(input_x, indices, updates)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]
  [1 1 1 1]]]
```

mindspore.ops.SearchSorted

class mindspore.ops.**SearchSorted**(*dtype=mstype.int64*, *right=False*)

Return the position indices where the elements can be inserted into the input tensor to maintain the increasing order of the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.searchsorted()* for more details.

Parameters

- **dtype** (*mindspore.dtype, optional*) –The specified type of output tensor. Optional values are: `mstype.int32` and `mstype.int64`. Default `mstype.int64`.
- **right** (*bool, optional*) –Search Strategy. If `True` , return the last suitable index found; if `False` , return the first such index. Default `False` .

Inputs:

- **sorted_sequence** (Tensor) - The input tensor. If *sorter* not provided, it must contain a increasing sequence on the innermost dimension.
- **values** (Tensor) - The value that need to be inserted.
- **sorter** (Tensor, optional) - An index sequence sorted in ascending order along the innermost dimension of *sorted_sequence* , which is used together with the unsorted *sorted_sequence* . CPU and GPU only support `None`. Default `None` .

Outputs:

Tensor

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> sorted_sequence = mindspore.tensor([1, 2, 2, 3, 4, 5, 5], mindspore.float32)
>>> values = mindspore.tensor([2], mindspore.float32)
>>> mindspore.ops.SearchSorted()(sorted_sequence, values)
Tensor(shape=[1], dtype=Int64, value= [1])
```

mindspore.ops.Select

class mindspore.ops.**Select**

```
prim = ops.Select()
out = prim(condition, input, other)
```

is equivalent to

```
ops.select(condition, input, other)
```

Refer to *mindspore.ops.select()* for more details.

mindspore.ops.Shape

class mindspore.ops.**Shape**

Returns the shape of the input tensor.

Refer to *mindspore.ops.shape()* for more details.

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

tuple[int], the output tuple is constructed by multiple integers, $(x_1, x_2, ..., x_R)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones(shape=[3, 2, 1]), mindspore.float32)
>>> shape = ops.Shape()
>>> output = shape(input_x)
>>> print(output)
(3, 2, 1)
```

mindspore.ops.Size

class mindspore.ops.**Size**

Returns a Scalar of type int that represents the size of the input Tensor and the total number of elements in the Tensor.

Refer to *mindspore.ops.size()* for more details.

Inputs:

- **input_x** (Tensor) - Input parameters, the shape of tensor is $(x_1, x_2, ..., x_R)$. The data type is number.

Outputs:

int. A scalar representing the elements' size of *input_x*, tensor is the number of elements in a tensor, $size = x_1 * x_2 * ...x_R$. The data type is an int.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[2, 2], [2, 2]]), mindspore.float32)
>>> size = ops.Size()
>>> output = size(input_x)
>>> print(output)
4
```

mindspore.ops.Slice

class mindspore.ops.**Slice**

Slices a tensor in the specified shape.

Refer to *mindspore.ops.slice()* for more details.

Inputs:

- **input_x** (Tensor): The target tensor. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.
- **begin** (Union[tuple, list]): The beginning of the slice. Only constant value(>=0) is allowed.
- **size** (Union[tuple, list]): The size of the slice. Only constant value is allowed.

Outputs:

Tensor, the shape is the same as that of *size*, the data type is the same as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import numpy as np
>>> data = Tensor(np.array([[[1, 1, 1], [2, 2, 2]],
...                         [[3, 3, 3], [4, 4, 4]],
...                         [[5, 5, 5], [6, 6, 6]]]).astype(np.int32))
>>> slice_op = ops.Slice()
>>> output = slice_op(data, (1, 0, 0), (1, 1, 3))
>>> print(output)
[[[3 3 3]]]
>>> output = slice_op(data, (1, 0, 0), (1, 1, 2))
>>> print(output)
[[[3 3]]]
>>> output = slice_op(data, (1, 0, 0), (1, 1, 1))
>>> print(output)
[[[3]]]
>>> output = slice_op(data, (1, 1, 0), (1, 1, 3))
>>> print(output)
[[[4 4 4]]]
>>> output = slice_op(data, (1, 0, 1), (1, 1, 2))
>>> print(output)
[[[3 3]]]
```

mindspore.ops.Sort

class mindspore.ops.**Sort**(*axis=-1*, *descending=False*)

Sorts the elements of the input tensor along the given dimension in the specified order.

> **Warning:** Currently, the data types of float16, uint8, int8, int16, int32, int64 are well supported. If use float32, it may cause loss of accuracy.

Parameters

- **axis** (*int, optional*) – The dimension to sort along. Default: `-1`, means the last dimension. The Ascend backend only supports sorting the last dimension.
- **descending** (*bool, optional*) – Controls the sort order. If descending is `True` then the elements are sorted in descending order by value. Default: `False` .

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

- **y1** (Tensor) - A tensor whose values are the sorted values, with the same shape and data type as input.
- **y2** (Tensor) - the indices of the elements in the original input tensor. Data type is int32.

Raises

- **TypeError** – If *axis* is not an int.
- **TypeError** – If *descending* is not a bool.
- **ValueError** – If *axis* is not in range of [-len(x.shape), len(x.shape)).

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[8, 2, 1], [5, 9, 3], [4, 6, 7]]), mindspore.float16)
>>> sort = ops.Sort()
>>> output = sort(x)
>>> # The output below is based on the Ascend platform.
>>> print(output)
(Tensor(shape=[3, 3], dtype=Float16, value=
[[ 1.0000e+00,  2.0000e+00,  8.0000e+00],
 [ 3.0000e+00,  5.0000e+00,  9.0000e+00],
 [ 4.0000e+00,  6.0000e+00,  7.0000e+00]]), Tensor(shape=[3, 3], dtype=Int32, value=
[[2, 1, 0],
 [2, 0, 1],
 [0, 1, 2]]))
```

mindspore.ops.SpaceToBatchND

class mindspore.ops.**SpaceToBatchND**(*block_shape*, *paddings*)

Divides spatial dimensions into blocks and combines the block size with the original batch.

This operation will divide spatial dimensions into blocks with *block_shape*, and then the output tensor's spatial dimension is the corresponding number of blocks after division. The output tensor's batch dimension is the product of the original batch and all elements in *block_shape*. Before division, the spatial dimensions of the input are zero padded according to paddings if necessary.

Parameters

- **block_shape** (*Union[list(int), tuple(int), int]*) – The block shape of dividing block with all elements greater than or euqal to 1. If *block_shape* is a list or tuple, the length of *block_shape* is the number of spatial dimensions, called M later. If *block_shape* is an int, the block size of M dimensions are the same, equal to *block_shape*. In this case of Ascend, M must be 2.
- **paddings** (*Union[tuple, list]*) – The padding values for spatial dimensions, containing M subtraction list. Each contains 2 integer values. All values must be greater than or equal to 0. *paddings[i]* specifies the paddings for the spatial dimension i, which corresponds to the input dimension i + offset,where offset = N-M, and N is the number of input dimensions. For each i, input_shape[i + offset]+paddings[i][0]+paddings[i][1] should be divisible by block_shape[i].

Inputs:

- **input_x** (Tensor) - The input tensor. The input tensor must be a 4-D tensor on Ascend.

Outputs:

Tensor, the output tensor with the same data type as the input. Assume the input shape is $(n, c_1, ...c_k, w_1, ..., w_M)$ with $block\_shape$ and $paddings$. The shape of the output tensor will be $(n', c_1, ...c_k, w'_1, ..., w'_M)$, where

$$n' = n * (block\_shape[0] * ... * block\_shape[M-1])$$
$$w'_i = (w_i + paddings[i-1][0] + paddings[i-1][1])//block\_shape[i-1]$$

Raises

- **TypeError** – If *block_shape* is not one of list, tuple, int.
- **TypeError** – If *paddings* is neither list nor tuple.
- **ValueError** – If *block_shape* is not one dimensional when *block_shape* is a list or tuple.
- **ValueError** – If the length of *block_shape* is not 2 on Ascend.
- **ValueError** – If shape of *paddings* is not (M, 2), where M is the length of *block_shape*.
- **ValueError** – If the element of *block_shape* is not an integer larger than or equal to 1.
- **ValueError** – If the element of *paddings* is not an integer larger than or euqal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> block_shape = [2, 2]
>>> paddings = [[0, 0], [0, 0]]
>>> space_to_batch_nd = ops.SpaceToBatchND(block_shape, paddings)
>>> input_x = Tensor(np.array([[[[1, 2], [3, 4]]]]), mindspore.float32)
>>> output = space_to_batch_nd(input_x)
>>> print(output)
[[[[1.]]]
 [[[2.]]]
 [[[3.]]]
 [[[4.]]]]
```

mindspore.ops.SpaceToDepth

class mindspore.ops.**SpaceToDepth**(*block_size*)

Rearrange blocks of spatial data into depth.

The output tensor's *height* dimension is $height/block\_size$.

The output tensor's *weight* dimension is $weight/block\_size$.

The depth of output tensor is $block\_size * block\_size * input\_depth$.

The input tensor's height and width must be divisible by *block_size*. The data format is "NCHW".

Parameters

block_size (*int*) – The block size used to divide spatial data. It must be >= 2.

Inputs:

- **x** (Tensor) - The target tensor. The data type is Number. It must be a 4-D tensor.

Outputs:

Tensor, the same data type as *x*. It must be a 4-D tensor. Tensor of shape $(N, (C_{in} * \text{block\_size} * 2), H_{in}/\text{block\_size}, W_{in}/\text{block\_size})$.

Raises

- **TypeError** – If *block_size* is not an int.
- **ValueError** – If *block_size* is less than 2.
- **ValueError** – If length of shape of *x* is not equal to 4.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.random.rand(1,3,2,2), mindspore.float32)
>>> block_size = 2
>>> space_to_depth = ops.SpaceToDepth(block_size)
>>> output = space_to_depth(x)
>>> print(output.shape)
(1, 12, 1, 1)
```

mindspore.ops.SparseGatherV2

class mindspore.ops.**SparseGatherV2**

Returns a slice of input tensor based on the specified indices and axis.

Inputs:

- **input_params** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.
- **input_indices** (Tensor) - The shape of tensor is $(y_1, y_2, ..., y_S)$. Specifies the indices of elements of the original Tensor, must be in the range *[0, input_params.shape[axis])*.
- **axis** (Union(int, Tensor[int])) - Specifies the dimension index to gather indices. When axis is Tensor, the size must be 1.

Outputs:

Tensor, the shape of tensor is $(z_1, z_2, ..., z_N)$.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_params = Tensor(np.array([[1, 2, 7, 42], [3, 4, 54, 22], [2, 2, 55, 3]]),␣
↪mindspore.float32)
>>> input_indices = Tensor(np.array([1, 2]), mindspore.int32)
>>> axis = 1
>>> out = ops.SparseGatherV2()(input_params, input_indices, axis)
>>> print(out)
[[2. 7.]
 [4. 54.]
 [2. 55.]]
```

mindspore.ops.Split

class mindspore.ops.**Split**(*axis=0*, *output_num=1*)

Splits the input tensor into output_num of tensors along the given axis and output numbers.

Refer to *mindspore.ops.split()* for more details.

Parameters

- **axis** (*int*) −Index of the split position. Default: `0` .
- **output_num** (*int*) −The number of output tensors. Must be positive int. Default: `1` .

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_0, x_1, ..., x_{R-1})$, R >= 1.

Outputs:

tuple[Tensor], the shape of each output tensor is the same, which is $(x_0, x_1, ..., x_{axis}/output\_num, ..., x_{R-1})$. And the data type is the same as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> split = ops.Split(1, 2)
>>> x = Tensor(np.array([[1, 1, 1, 1], [2, 2, 2, 2]]), mindspore.int32)
>>> print(x)
[[1 1 1 1]
 [2 2 2 2]]
>>> output = split(x)
>>> print(output)
(Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 1],
 [2, 2]]), Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 1],
```

(continues on next page)

(continued from previous page)

```
 [2, 2]]))
>>> split = ops.Split(1, 4)
>>> output = split(x)
>>> print(output)
(Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [2]]), Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [2]]), Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [2]]), Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [2]]))
```

mindspore.ops.Squeeze

class mindspore.ops.**Squeeze**(*axis=()*)

Return the Tensor after deleting the dimension of size 1 in the specified *axis*.

Refer to *mindspore.ops.squeeze()* for more details.

> **Warning:** This is an experimental API that is subject to change or deletion.

Parameters

axis (*Union[int, tuple(int)]*) – Specifies the dimension indexes of shape to be removed, which will remove all the dimensions of size 1 in the given axis parameter. If specified, it must be int32 or int64. Default: `()` .

Inputs:

- **input** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor, the shape of tensor is $(x_1, x_2, ..., x_S)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input = Tensor(np.ones(shape=[3, 2, 1]), mindspore.float32)
>>> squeeze = ops.Squeeze(2)
>>> output = squeeze(input)
>>> print(output)
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

mindspore.ops.Stack

class mindspore.ops.**Stack**(*axis=0*)

Stacks a list of tensors in specified axis.

Refer to *mindspore.ops.stack()* for more details.

Parameters

axis (*int, optional*) – Dimension to stack. The range is [-(R+1), R+1). Default: `0` .

Inputs:

- **input_x** (Union[tuple, list]) - A Tuple or list of Tensor objects with the same shape and type.

Outputs:

Tensor. A stacked Tensor with the same type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> import numpy as np
>>> data1 = Tensor(np.array([0, 1]).astype(np.float32))
>>> data2 = Tensor(np.array([2, 3]).astype(np.float32))
>>> stack = ops.Stack()
>>> output = stack([data1, data2])
>>> print(output)
[[0. 1.]
 [2. 3.]]
```

mindspore.ops.StridedSlice

class mindspore.ops.**StridedSlice**(*begin_mask=0*, *end_mask=0*, *ellipsis_mask=0*, *new_axis_mask=0*, *shrink_axis_mask=0*)

```
prim = ops.StridedSlice(begin_mask, end_mask, ellipsis_mask, new_axis_mask, shrink_axis_mask)
out = prim(input_x, begin, end, strides)
```

is equivalent to

```
ops.strided_slice(input_x, begin, end, strides, begin_mask, end_mask, ellipsis_mask, new_
↪axis_mask, shrink_axis_mask)
```

Refer to *mindspore.ops.strided_slice()* for more details.

mindspore.ops.TensorScatterAdd

class mindspore.ops.**TensorScatterAdd**

Creates a new tensor by adding the values from the positions in *input_x* indicated by *indices*, with values from *updates*. When multiple values are given for the same index, the updated result will be the sum of all values. This operation is almost equivalent to using *mindspore.ops.ScatterNdAdd*, except that the updates are applied on output *Tensor* instead of input *Parameter*.

Refer to *mindspore.ops.tensor_scatter_add()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as input, and updates. Shape should be equal to indices.shape[:-1] + input_x.shape[indices.shape[-1]:].

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterAdd()
>>> # 5, Perform the addition operation for the first time:
>>> #      first_input_x = input_x[0][0] + updates[0] = [[0.9, 0.3, 3.6], [0.4, 0.5, -3.2]]
>>> # 6, Perform the addition operation for the second time:
>>> #      second_input_x = input_x[0][0] + updates[1] = [[3.1, 0.3, 3.6], [0.4, 0.5, -3.2]]
>>> output = op(input_x, indices, updates)
>>> print(output)
[[ 3.1  0.3  3.6]
 [ 0.4  0.5 -3.2]]
```

mindspore.ops.TensorScatterDiv

class mindspore.ops.**TensorScatterDiv**

Creates a new tensor by dividing the values from the positions in *input_x* indicated by *indices*, with values from *updates*. When divided values are provided for the same index, the result of the update will be to divided these values respectively. Except that the updates are applied on output *Tensor* instead of input *Parameter*.

Refer to *mindspore.ops.tensor_scatter_div()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as input, and updates.shape should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.0]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterDiv()
>>> # 5, Perform the division operation for the first time:
>>> #      first_input_x = input_x[0][0] / updates[0] = [[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]
>>> # 6, Perform the division operation for the second time:
>>> #      second_input_x = input_x[0][0] * updates[1] = [[-0.05, 0.3, 3.6], [0.4, 0.5, -3.
 ↪2]]
>>> output = op(input_x, indices, updates)
>>> print(output)
[[-0.05  0.3  3.6  ]
 [ 0.4   0.5  -3.2 ]]
```

mindspore.ops.TensorScatterMax

class mindspore.ops.**TensorScatterMax**

By comparing the value at the position indicated by *indices* in *x* with the value in the *updates*, the value at the index will eventually be equal to the largest one to create a new tensor.

Refer to *mindspore.ops.tensor_scatter_max()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as input, and updates.shape should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterMax()
>>> # 5, Perform the max operation for the first time:
>>> #      first_input_x = Max(input_x[0][0], updates[0]) = [[1.0, 0.3, 3.6], [0.4, 0.5, -3.
 ↪2]]
>>> # 6, Perform the max operation for the second time:
>>> #      second_input_x = Max(input_x[0][0], updates[1]) = [[2.2, 0.3, 3.6], [0.4, 0.5, -3.
 ↪2]]
>>> output = op(input_x, indices, updates)
>>> print(output)
[[ 2.2  0.3  3.6]
 [ 0.4  0.5 -3.2]]
```

mindspore.ops.TensorScatterMin

class mindspore.ops.**TensorScatterMin**

By comparing the value at the position indicated by *indices* in *input_x* with the value in the *updates*, the value at the index will eventually be equal to the smallest one to create a new tensor.

Refer to *mindspore.ops.tensor_scatter_min()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as input, and updates.shape should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterMin()
>>> # 5, Perform the min operation for the first time:
>>> #      first_input_x = Min(input_x[0][0], updates[0]) = [[-0.1, 0.3, 3.6], [0.4, 0.5, -3.
↪2]]
>>> # 6, Perform the min operation for the second time:
>>> #      second_input_x = Min(input_x[0][0], updates[1]) = [[-0.1, 0.3, 3.6], [0.4, 0.5, -
↪3.2]]
>>> output = op(input_x, indices, updates)
>>> print(output)
[[ -0.1  0.3  3.6]
 [ 0.4  0.5 -3.2]]
```

mindspore.ops.TensorScatterMul

class mindspore.ops.**TensorScatterMul**

Creates a new tensor by multiplying the values from the positions in *input_x* indicated by *indices*, with values from *updates*. When multiple values are provided for the same index, the result of the update will be to multiply these values respectively. The updates are applied on output *Tensor* instead of input *Parameter*.

$$output\left[indices\right] = input\_x \times update$$

Refer to *mindspore.ops.tensor_scatter_mul()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as *input_x*, and the shape of *updates* should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterMul()
>>> # 5, Perform the multiply operation for the first time:
>>> #      first_input_x = input_x[0][0] * updates[0] = [[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]
>>> # 6, Perform the multiply operation for the second time:
>>> #      second_input_x = input_x[0][0] * updates[1] = [[-0.22, 0.3, 3.6], [0.4, 0.5, -3.
 ↪2]]
>>> output = op(input_x, indices, updates)
>>> print(output)
[[-0.22  0.3   3.6  ]
 [ 0.4   0.5   -3.2 ]]
```

mindspore.ops.TensorScatterSub

class mindspore.ops.**TensorScatterSub**

Creates a new tensor by subtracting the values from the positions in *input_x* indicated by *indices*, with values from *updates*. When multiple values are provided for the same index, the result of the update will subtract these values respectively. This operation is almost equivalent to using *mindspore.ops.ScatterNdSub* , except that the updates are applied on output *Tensor* instead of input *Parameter*.

```
# Iterate through all index
for i in range(indices.shape[0]):
    for j in range(indices.shape[1]):
        ...
        for k in range(indices.shape[-2]):  # The last dimension is coordinate dimension
            # Get current index combination
            index_tuple = (i, j, ..., k)
            # Get target position
            target_index = indices[index_tuple]
            # Get corresponding update value
            update_value = updates[index_tuple]
            # Perform subtraction operation
            output[target_index] -= update_value
```

Refer to *mindspore.ops.tensor_scatter_sub()* for more details.

Inputs:

- **input_x** (Tensor) - The target tensor. The dimension of input_x must be no less than indices.shape[-1].
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **updates** (Tensor) - The tensor to update the input tensor, has the same type as *input_x*, and the shape of *updates* should be equal to $indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$.

Outputs:

Tensor, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [0, 0]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> # Next, demonstrate the approximate operation process of this operator:
>>> # 1, indices[0] = [0, 0], indices[1] = [0, 0]
>>> # 2, And input_x[0, 0] = -0.1
>>> # 3, So input_x[indices] = [-0.1, -0.1]
>>> # 4, Satisfy the above formula: input_x[indices].shape=(2) == updates.shape=(2)
>>> op = ops.TensorScatterSub()
>>> # 5, Perform the subtract operation for the first time:
>>> #      first_input_x = input_x[0][0] - updates[0] = [[-1.1, 0.3, 3.6], [0.4, 0.5, -3.2]]
>>> # 6, Perform the subtract operation for the second time:
>>> #      second_input_x = input_x[0][0] - updates[1] = [[-3.3, 0.3, 3.6], [0.4, 0.5, -3.2]]
```

(continues on next page)

(continued from previous page)

```
>>> output = op(input_x, indices, updates)
>>> print(output)
[[-3.3000002  0.3        3.6      ]
 [ 0.4        0.5       -3.2      ]]
```

mindspore.ops.TensorScatterUpdate

class mindspore.ops.**TensorScatterUpdate**

Creates a new tensor by updating the positions in *input_x* indicated by *indices*, with values from *update*. This operation is almost equivalent to using *mindspore.ops.ScatterNdUpdate* , except that the updates are applied on output *Tensor* instead of *input_x*.

indices must have rank at least 2, the last axis is the depth of each index vectors. For each index vector, there must be a corresponding value in *update*. If the depth of each index tensor matches the rank of *input_x*, then each index vector corresponds to a scalar in *input_x* and each *update* updates a scalar. If the depth of each index tensor is less than the rank of *input_x*, then each index vector corresponds to a slice in *input_x*, and each *update* updates a slice.

The order in which updates are applied is nondeterministic, meaning that if there are multiple index vectors in *indices* that correspond to the same position, the value of that position in the output will be nondeterministic.

$$output[indices] = update$$

Inputs:

- **input_x** (Tensor) - The input tensor. The dimension of input_x must be no less than indices.shape[-1]. The shape is $(N, *)$ where $*$ means,any number of additional dimensions. The data type is Number.
- **indices** (Tensor) - The index of input tensor whose data type is int32 or int64. The rank must be at least 2.
- **update** (Tensor) - The tensor to update the input tensor, has the same type as *input_x*, and $update.shape = indices.shape[:-1] + input\_x.shape[indices.shape[-1]:]$

Outputs:

Tensor, has the same shape and type as *input_x*.

Raises

- **TypeError** –If dtype of *indices* is neither int32 nor int64.
- **ValueError** –If length of shape of *input_x* is less than the last dimension of shape of *indices*.
- **ValueError** –If the value of *input_x* are not match with input *indices*.
- **RuntimeError** –If a value of *indices* is not in *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]]), mindspore.float32)
>>> indices = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> update = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> op = ops.TensorScatterUpdate()
>>> output = op(input_x, indices, update)
>>> print(output)
[[ 1.   0.3  3.6]
 [ 0.4  2.2 -3.2]]
```

mindspore.ops.TensorShape

class mindspore.ops.**TensorShape**

Returns the shape of the input tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> input_x = Tensor(np.ones(shape=[3, 2, 1]), mindspore.float32)
>>> output = ops.TensorShape()(input_x)
>>> print(output)
[3 2 1]
```

mindspore.ops.Tile

class mindspore.ops.**Tile**

Replicates an input tensor with given multiple times.

Refer to *mindspore.ops.tile()* for more details.

Inputs:

- **input** (Tensor) - The tensor whose elements need to be repeated. Set the shape of input tensor as $(x_1, x_2, ..., x_S)$.
- **dims** (tuple[int]) - The parameter that specifies the number of replications, the parameter type is tuple, and the data type is int, i.e., $(y_1, y_2, ..., y_S)$. Only constant value is allowed.

Note: On Ascend, the number of *dims* should not exceed 8, and currently does not support scenarios where more than 4 dimensions are repeated simultaneously.

Outputs:

Tensor, has the same data type as the *input*. Suppose the length of *dims* is d, the dimension of *input* is *input.dim*, and the shape of *input* is $(x_1, x_2, ..., x_S)$.

- If *input.dim* = d, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * y_1, x_2 * y_2, ..., x_S * y_S)$.
- If *input.dim* < d, prepend 1 to the shape of *input* until their lengths are consistent. Such as set the shape of *input* as $(1, ..., x_1, x_2, ..., x_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(1 * y_1, ..., x_R * y_R, x_S * y_S)$.
- If *input.dim* > d, prepend 1 to *dims* until their lengths are consistent. Such as set the *dims* as $(1, ..., y_1, y_2, ..., y_S)$, then the shape of their corresponding positions can be multiplied, and the shape of Outputs is $(x_1 * 1, ..., x_R * y_R, x_S * y_S)$.

Raises

- **TypeError** – If *dims* is not a tuple or its elements are not all int.
- **ValueError** – If the elements of *dims* are not all greater than or equal to 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> tile = ops.Tile()
>>> input = Tensor(np.array([[1, 2], [3, 4]]), mindspore.float32)
>>> dims = (2, 3)
>>> output = tile(input, dims)
>>> print(output)
[[1.  2.  1.  2.  1.  2.]
 [3.  4.  3.  4.  3.  4.]
 [1.  2.  1.  2.  1.  2.]
 [3.  4.  3.  4.  3.  4.]]
>>> dims = (2, 3, 2)
>>> output = tile(input, dims)
>>> print(output)
[[[1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]]
 [[1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]
  [1. 2. 1. 2.]
  [3. 4. 3. 4.]]]
```

mindspore.ops.Trace

class mindspore.ops.**Trace**

```
prim = ops.Trace()
out = prim(input)
```

is equivalent to

```
ops.trace(input)
```

Refer to *mindspore.ops.trace()* for more details.

mindspore.ops.Transpose

class mindspore.ops.**Transpose**

```
prim = ops.Transpose()
out = prim(input, input_perm)
```

is equivalent to

```
ops.transpose(input, input_perm)
```

Refer to *mindspore.ops.transpose()* for more details.

mindspore.ops.Tril

class mindspore.ops.**Tril**(*diagonal=0*)

Returns the lower triangular portion of the 2-D matrix or the set of matrices in a batch. The remaining elements of the resulting Tensor are assigned a value of 0. The lower triangular section of the matrix comprises of the elements present on and below the main diagonal.

Warning: This is an experimental API that is subject to change or deletion.

Parameters

diagonal (*int, optional*) – An optional attribute indicates the diagonal to consider, default: `0` , indicating the main diagonal.

Inputs:

- **x** (Tensor) - The input tensor with shape $(*, M, N)$ where $*$ means any number of additional dimensions.

Outputs:

Tensor, the same shape and data type as the input *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If *diagonal* is not an int.
- **ValueError** – If the rank of *x* is less than 2.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> x = Tensor(np.array([[ 1,  2,  3,  4],
...                      [ 5,  6,  7,  8],
...                      [10, 11, 12, 13],
...                      [14, 15, 16, 17]]))
>>> tril = ops.Tril()
>>> result = tril(x)
>>> print(result)
[[ 1  0  0  0]
 [ 5  6  0  0]
 [10 11 12  0]
 [14 15 16 17]]
>>> x = Tensor(np.array([[ 1,  2,  3,  4],
...                      [ 5,  6,  7,  8],
...                      [10, 11, 12, 13],
...                      [14, 15, 16, 17]]))
>>> tril = ops.Tril(diagonal=1)
>>> result = tril(x)
>>> print(result)
[[ 1  2  0  0]
 [ 5  6  7  0]
 [10 11 12 13]
 [14 15 16 17]]
>>> x = Tensor(np.array([[ 1,  2,  3,  4],
...                      [ 5,  6,  7,  8],
...                      [10, 11, 12, 13],
...                      [14, 15, 16, 17]]))
>>> tril = ops.Tril(diagonal=-1)
>>> result = tril(x)
>>> print(result)
[[ 0  0  0  0]
 [ 5  0  0  0]
 [10 11  0  0]
 [14 15 16  0]]
```

mindspore.ops.TrilIndices

class mindspore.ops.**TrilIndices**(*row*, *col*, *offset=0*, *dtype=mstype.int32*)

Computes the indices of the lower triangular elements of a 2D matrix and returns them as a Tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.tril_indices()* for more details.

Parameters

- **row** (*int*) –number of rows in the 2-D matrix.

- **col** (*int*) –number of columns in the 2-D matrix.
- **offset** (*int, optional*) –diagonal offset from the main diagonal. Default: `0` .
- **dtype** (*mindspore.dtype*, optional) –The specified type of output tensor. An optional data type of `mstype.int32` and `mstype.int64` . Default: `mstype.int32` .

Outputs:

- **y** (Tensor) - indices of the elements in lower triangular part of matrix. The type specified by *dtype*. The shape of output is $(2, tril\_size)$, where $tril\_size$ is the number of elements in the lower triangular matrix.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> from mindspore import dtype as mstype
>>> net = ops.TrilIndices(4, 3, -1, mstype.int64)
>>> output = net()
>>> print(output)
[[1 2 2 3 3 3]
 [0 0 1 0 1 2]]
>>> print(output.dtype)
Int64
```

mindspore.ops.Triu

class mindspore.ops.**Triu**(*diagonal=0*)

```
prim = ops.Triu(diagonal)
out = prim(input)
```

is equivalent to

```
ops.triu(input, diagonal)
```

Refer to *mindspore.ops.triu()* for more details.

mindspore.ops.TriuIndices

class mindspore.ops.**TriuIndices**(*row*, *col*, *offset=0*, *dtype=mstype.int32*)

Calculates the indices of the upper triangular elements in a *row* * *col* matrix and returns them as a 2-by-N Tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.triu_indices()* for more details.

Parameters

- **row** (*int*) –number of rows in the 2-D matrix.
- **col** (*int*) –number of columns in the 2-D matrix.

- **offset** (*int, optional*) –diagonal offset from the main diagonal. Default: `0` .
- **dtype** (*mindspore.dtype*, optional) –The specified type of output tensor. An optional data type of `mstype.int32` and `mstype.int64` . Default: `mstype.int32` .

Outputs:

- **y** (Tensor) - indices of the elements in upper triangular part of matrix. The type specified by *dtype*. The shape of output is $(2, tril\_size)$, where $tril\_size$ is the number of elements in the upper triangular matrix.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> from mindspore import dtype as mstype
>>> net = ops.TriuIndices(5, 4, 2, mstype.int64)
>>> output = net()
>>> print(output)
[[0 0 1]
 [2 3 3]]
>>> print(output.dtype)
Int64
```

mindspore.ops.Unique

class mindspore.ops.**Unique**

Returns the unique elements of input tensor and also return a tensor containing the index of each value of input tensor corresponding to the output unique tensor.

The output contains Tensor *y* and Tensor *idx*, the format is probably similar to (*y*, *idx*). The shape of Tensor *y* and Tensor *idx* is different in most cases, because Tensor *y* will be duplicated, and the shape of Tensor *idx* is consistent with the input.

To get the same shape between *idx* and *y*, please refer to *mindspore.ops.UniqueWithPad*.

Inputs:

- **input_x** (Tensor) - The input tensor. The shape is $(N, *)$ where $*$ means, any number of additional dimensions.

Outputs:

Tuple, containing Tensor objects (*y*, *idx*), *y* is a tensor with the same type as *input_x*, and contains the unique elements in *x*. *idx* is a tensor containing indices of elements in the input corresponding to the output tensor.

Raises

TypeError –If *input_x* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, nn
>>> input_x = Tensor(np.array([1, 2, 5, 2]), mindspore.int32)
>>> output = ops.Unique()(input_x)
>>> print(output)
(Tensor(shape=[3], dtype=Int32, value= [1, 2, 5]), Tensor(shape=[4], dtype=Int32, value= [0,␣
↪1, 2, 1]))
>>> y = output[0]
>>> print(y)
[1 2 5]
>>> idx = output[1]
>>> print(idx)
[0 1 2 1]
>>> # As can be seen from the above, y and idx shape
>>> # note that for GPU, this operator must be wrapped inside a model, and executed in graph␣
↪mode.
>>> class UniqueNet(nn.Cell):
...     def __init__(self):
...         super(UniqueNet, self).__init__()
...         self.unique_op = ops.Unique()
...
...     def construct(self, x):
...         output, indices = self.unique_op(x)
...         return output, indices
...
>>> input_x = Tensor(np.array([1, 2, 5, 2]), mindspore.int32)
>>> net = UniqueNet()
>>> output = net(input_x)
>>> print(output)
(Tensor(shape=[3], dtype=Int32, value= [1, 2, 5]), Tensor(shape=[4], dtype=Int32, value= [0,␣
↪1, 2, 1]))
```

mindspore.ops.UniqueConsecutive

class mindspore.ops.**UniqueConsecutive**(*return_inverse=False*, *return_counts=False*, *dim=None*)

Returns the elements that are unique in each consecutive group of equivalent elements in the input tensor.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.unique_consecutive()* for more details.

Parameters

- **return_inverse** (*bool, optional*) – Whether to return the index of where the element in the original input maps to the position in the output. Default: `False` .
- **return_counts** (*bool, optional*) – Whether to return the counts of each unique element. Default: `False` .
- **dim** (*int, optional*) – The dimension to apply unique. If `None` , the unique of the flattened input is returned. If specified, it must be int32 or int64. Default: `None` .

Inputs:

- **x** (Tensor) - The input tensor.

Outputs:

A tensor or a tuple of tensors containing tensor objects (*output*, *idx*, *counts*).

- *output* has the same type as *x* and is used to represent the output list of unique scalar elements.
- If *return_inverse* is True, there will be an additional returned tensor, *idx*, which has the same shape as *x* and represents the index of where the element in the original input maps to the position in the output.
- If *return_counts* is True, there will be an additional returned tensor, *counts*, which represents the number of occurrences for each unique value or tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> x = Tensor(np.array([1, 1, 2, 2, 3, 1, 1, 2]), mstype.int32)
>>> unique_consecutive = ops.UniqueConsecutive(True, True, None)
>>> output, idx, counts = unique_consecutive(x)
>>> print(output)
[1 2 3 1 2]
>>> print(idx)
[0 0 1 1 2 3 3 4]
>>> print(counts)
[2 2 1 2 1]
```

mindspore.ops.UniqueWithPad

class mindspore.ops.**UniqueWithPad**

'ops.UniqueWithPad' is deprecated from version 2.4 and will be removed in a future version. Please use the *mindspore.ops.unique()* combined with *mindspore.ops.pad()* to realize the same function.

Supported Platforms:

Deprecated

mindspore.ops.UnsortedSegmentMax

class mindspore.ops.**UnsortedSegmentMax**

Computes the maximum along segments of a tensor.

Refer to *mindspore.ops.unsorted_segment_max()* for more details.

Inputs:

- **input_x** (Tensor) - The shape is $(x_1, x_2, ..., x_R)$. The data type must be float16, float32 or int32.
- **segment_ids** (Tensor) - The label indicates the segment to which each element belongs. Set the shape as $(x_1, x_2, ..., x_N)$, where 0 < N <= R.
- **num_segments** (Union[int, Tensor]) - Set z as num_segments, it can be an int or 0-D Tensor.

Outputs:

Tensor, the shape is $(z, x_{N+1}, ..., x_R)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> # case 1: Only have two num_segments, where is 0 and 1, and segment_ids=[0, 1, 1]
>>> # num_segments = 2 indicates that there are two types of segment_id,
>>> # the first number '0' in [0, 1, 1] indicates input_x[0],
>>> # the second number '1' in [0, 1, 1] indicates input_x[1],
>>> # the third number '1' in [0, 1, 1] indicates input_x[2],
>>> # input_x[0], which is [1, 2, 3] will not be compared to other segment_id.
>>> # Only the same segment_id will be compared.
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import numpy as np
>>> input_x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [4, 2, 1]]).astype(np.float32))
>>> segment_ids = Tensor(np.array([0, 1, 1]).astype(np.int32))
>>> num_segments = 2
>>> unsorted_segment_max = ops.UnsortedSegmentMax()
>>> output = unsorted_segment_max(input_x, segment_ids, num_segments)
>>> print(output)
[[1. 2. 3.]
 [4. 5. 6.]]
>>>
>>> # case 2: The segment_ids=[0, 0, 1, 1].
>>> # [1, 2, 3] will compare with [4, 2, 0],
>>> # and [4, 5, 6] will compare with [4, 2, 1].
>>> input_x = Tensor(np.array([[1, 2, 3], [4, 2, 0], [4, 5, 6], [4, 2, 1]]).astype(np.
↪float32))
>>> segment_ids = Tensor(np.array([0, 0, 1, 1]).astype(np.int32))
>>> num_segments = 2
>>> unsorted_segment_max = ops.UnsortedSegmentMax()
>>> output = unsorted_segment_max(input_x, segment_ids, num_segments)
>>> print(input_x.shape)
    (4, 3)
>>> print(output)
    [[4. 2. 3.]
     [4. 5. 6.]]
>>> # case 3: If the input_x have three dimensions even more, what will happen?
>>> # The shape of input_x is (2, 4, 3),
>>> # and the length of segment_ids should be the same as the first dimension of input_x.
>>> # Because the segment_ids are different, input_x[0] will not be compared to input_x[1].
>>> input_x = Tensor(np.array([[[1, 2, 3], [4, 2, 0], [4, 5, 6], [4, 2, 1]],
...                            [[1, 2, 3], [4, 2, 0], [4, 5, 6], [4, 2, 1]]]).astype(np.
↪float32))
>>> segment_ids = Tensor(np.array([0, 1]).astype(np.int32))
>>> num_segments = 2
>>> unsorted_segment_max = ops.UnsortedSegmentMax()
>>> output = unsorted_segment_max(input_x, segment_ids, num_segments)
>>> print(input_x.shape)
    (2, 4, 3)
>>> print(output)
    [[[1. 2. 3.]
```

(continues on next page)

(continued from previous page)

```
      [4. 2. 0.]
      [4. 5. 6.]
      [4. 2. 1.]]
     [[1. 2. 3.]
      [4. 2. 0.]
      [4. 5. 6.]
      [4. 2. 1.]]]
>>> # case 4: It has the same input with the 3rd case.
>>> # Because num_segments is equal to 2, there are two segment_ids, but currently only one␣
 ↪0 is used.
>>> # the segment_id i is absent in the segment_ids, then output[i] will be filled with
>>> # the smallest possible value of the input_x's type.
>>> segment_ids = Tensor(np.array([0, 0]).astype(np.int32))
>>> output = unsorted_segment_max(input_x, segment_ids, num_segments)
>>> print(output)
    [[[ 1.0000000e+00  2.0000000e+00  3.0000000e+00]
      [ 4.0000000e+00  2.0000000e+00  0.0000000e+00]
      [ 4.0000000e+00  5.0000000e+00  6.0000000e+00]
      [ 4.0000000e+00  2.0000000e+00  1.0000000e+00]]
     [[-3.4028235e+38 -3.4028235e+38 -3.4028235e+38]
      [-3.4028235e+38 -3.4028235e+38 -3.4028235e+38]
      [-3.4028235e+38 -3.4028235e+38 -3.4028235e+38]
      [-3.4028235e+38 -3.4028235e+38 -3.4028235e+38]]]
```

mindspore.ops.UnsortedSegmentMin

class mindspore.ops.**UnsortedSegmentMin**

Computes the minimum of a tensor along segments.

Refer to *mindspore.ops.unsorted_segment_min()* for more details.

Inputs:

- **input_x** (Tensor) - The shape is $(x_1, x_2, ..., x_R)$. The data type must be float16, float32 or int32.
- **segment_ids** (Tensor) - The label indicates the segment to which each element belongs. Set the shape as $(x_1, x_2, ..., x_N)$, where 0 < N <= R.
- **num_segments** (Union[int, Tensor]) - Set z as num_segments, it can be an int or 0-D Tensor.

Outputs:

Tensor, the shape is $(z, x_{N+1}, ..., x_R)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import numpy as np
>>> input_x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [4, 2, 1]]).astype(np.float32))
>>> segment_ids = Tensor(np.array([0, 1, 1]).astype(np.int32))
>>> num_segments = 2
>>> unsorted_segment_min = ops.UnsortedSegmentMin()
>>> output = unsorted_segment_min(input_x, segment_ids, num_segments)
>>> print(output)
[[1. 2. 3.]
 [4. 2. 1.]]
```

mindspore.ops.UnsortedSegmentProd

class mindspore.ops.**UnsortedSegmentProd**

Computes the product of a tensor along segments.

Refer to *mindspore.ops.unsorted_segment_prod()* for more details.

Inputs:

- **input_x** (Tensor) - The shape is $(x_1, x_2, ..., x_R)$. With float16, float32 or int32 data type.
- **segment_ids** (Tensor) - The label indicates the segment to which each element belongs. Set the shape as $(x_1, x_2, ..., x_N)$, where 0 < N <= R. Data type must be int32.
- **num_segments** (Union[int, Tensor]) - Set z as num_segments, it can be an int or 0-D Tensor.

Outputs:

Tensor, the shape is $(z, x_{N+1}, ..., x_R)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> import numpy as np
>>> input_x = Tensor(np.array([[1, 2, 3], [4, 5, 6], [4, 2, 1]]).astype(np.float32))
>>> segment_ids = Tensor(np.array([0, 1, 0]).astype(np.int32))
>>> num_segments = 2
>>> unsorted_segment_prod = ops.UnsortedSegmentProd()
>>> output = unsorted_segment_prod(input_x, segment_ids, num_segments)
>>> print(output)
[[4. 4. 3.]
 [4. 5. 6.]]
```

mindspore.ops.UnsortedSegmentSum

class mindspore.ops.**UnsortedSegmentSum**

```
prim = ops.UnsortedSegmentSum()
out = prim(input_x, segment_ids, num_segments)
```

is equivalent to

```
ops.unsorted_segment_sum(input_x, segment_ids, num_segments)
```

Refer to *mindspore.ops.unsorted_segment_sum()* for more details.

mindspore.ops.Unstack

class mindspore.ops.**Unstack**(*axis=0*, *num=None*)

Unstacks tensor in specified axis, this is the opposite of ops.Stack. Assuming input is a tensor of rank *R*, output tensors will have rank *(R-1)*.

Refer to *mindspore.ops.unstack()* for more details.

Parameters

- **axis** (*int, optional*) –Dimension along which to unpack. Default: `0` . Negative values wrap around. The range is [-R, R).
- **num** (*Union[None, int], optional*) –The number of output tensors. Automatically inferred by input_x and axis if `None` . Default: `None` .

Inputs:

- **input_x** (Tensor) - The shape is $(x_1, x_2, ..., x_R)$. A tensor to be unstacked and the rank of the tensor must be greater than 0.

Outputs:

A tuple of tensors, the shape of each objects is the same. Given a tensor of shape $(x_1, x_2, ..., x_R)$. If $0 \leq axis$, the shape of tensor in output is $(x_1, x_2, ..., x_{axis}, x_{axis+2}, ..., x_R)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> unstack = ops.Unstack()
>>> input_x = Tensor(np.array([[1, 1, 1, 1], [2, 2, 2, 2]]))
>>> output = unstack(input_x)
>>> print(output)
(Tensor(shape=[4], dtype=Int64, value= [1, 1, 1, 1]), Tensor(shape=[4], dtype=Int64, value=␣
↪[2, 2, 2, 2]))
```

18.4.4 Type Conversion

API Name	Description	Supported Platforms
mindspore.ops.ScalarCast	'ops.ScalarCast' is deprecated from version 2.3 and will be removed in a future version, please use *int(x)* or *float(x)* instead.	Deprecated
mindspore.ops.ScalarToTensor	Converts a scalar to a *Tensor*, and converts the data type to the specified type.	Ascend GPU CPU
mindspore.ops.TupleToArray	Converts a tuple to a tensor.	Ascend GPU CPU

mindspore.ops.ScalarCast

class mindspore.ops.**ScalarCast**

'ops.ScalarCast' is deprecated from version 2.3 and will be removed in a future version, please use *int(x)* or *float(x)* instead.

Supported Platforms:

Deprecated

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> scalar_cast = ops.ScalarCast()
>>> output = scalar_cast(255.0, mindspore.int64)
>>> print(output)
255
```

mindspore.ops.ScalarToTensor

class mindspore.ops.**ScalarToTensor**

Converts a scalar to a *Tensor*, and converts the data type to the specified type.

Refer to *mindspore.ops.scalar_to_tensor()* for more details.

Inputs:

- **input_x** (Union[int, float]) - The input is a scalar. Only constant value is allowed.
- **dtype** (mindspore.dtype) - The target data type. Default: `mindspore.float32` . Only constant value is allowed.

Outputs:

Tensor. 0-D Tensor and the content is the input.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> op = ops.ScalarToTensor()
>>> data = 1
>>> output = op(data, mindspore.float32)
>>> print(output)
1.0
```

mindspore.ops.TupleToArray

class mindspore.ops.**TupleToArray**

Converts a tuple to a tensor.

Refer to *mindspore.ops.tuple_to_array()* for more details.

Inputs:

- **input_x** (tuple) - A tuple of numbers. These numbers have the same type. The shape is $(N,)$.

Outputs:

Tensor, if the input tuple contains N numbers, then the shape of the output tensor is $(N,)$.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> input_x = (1,2,3)
>>> print(type(input_x))
<class 'tuple'>
>>> output = ops.TupleToArray()(input_x)
>>> print(type(output))
<class 'mindspore.common.tensor.Tensor'>
>>> print(output)
[1 2 3]
```

18.5 Parameter Operation Operator

API Name	Description	Supported Platforms
mindspore.ops.Assign	Refer to *mindspore.ops.assign()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.AssignAdd	Refer to *mindspore.ops.assign_add()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.AssignSub	Refer to *mindspore.ops.assign_sub()* for more details.	`Ascend` `GPU` `CPU`
mindspore.ops.ScatterAdd	Updates the value of the input tensor through the addition operation.	`Ascend` `GPU` `CPU`

continues on next page

Table 19 – continued from previous page

mindspore.ops.ScatterDiv	Updates the value of the input tensor through the divide operation.	Ascend GPU CPU
mindspore.ops.ScatterMax	Updates the value of the input tensor through the maximum operation.	Ascend GPU CPU
mindspore.ops.ScatterMin	Updates the value of the input tensor through the minimum operation.	Ascend GPU CPU
mindspore.ops.ScatterMul	Updates the value of the input tensor through the multiply operation.	Ascend GPU CPU
mindspore.ops.ScatterNdAdd	Applies sparse addition to individual values or slices in a tensor.	Ascend GPU CPU
mindspore.ops.ScatterNdSub	Applies sparse subtraction to individual values or slices in a tensor.	Ascend GPU CPU
mindspore.ops.ScatterNdUpdate	Updates tensor values by using input indices and value.	Ascend GPU CPU
mindspore.ops.ScatterNonAliasingAdd	Please use TensorScatterAdd instead.	Deprecated
mindspore.ops.ScatterSub	Updates the value of the input tensor through the subtraction operation.	Ascend GPU CPU
mindspore.ops.ScatterUpdate	Updates tensor values by using input indices and value.	Ascend GPU CPU

18.5.1 mindspore.ops.Assign

class mindspore.ops.**Assign**

```
prim = ops.Assign()
out = prim(variable, value)
```

is equivalent to

```
ops.assign(variable, value)
```

Refer to *mindspore.ops.assign()* for more details.

18.5.2 mindspore.ops.AssignAdd

class mindspore.ops.**AssignAdd**

```
prim = ops.AssignAdd()
out = prim(variable, value)
```

is equivalent to

```
ops.assign_add(variable, value)
```

Refer to *mindspore.ops.assign_add()* for more details.

18.5.3 mindspore.ops.AssignSub

class mindspore.ops.**AssignSub**

```
prim = ops.AssignSub()
out = prim(variable, value)
```

is equivalent to

```
ops.assign_sub(variable, value)
```

Refer to *mindspore.ops.assign_sub()* for more details.

18.5.4 mindspore.ops.ScatterAdd

class mindspore.ops.**ScatterAdd**(*use_locking=False*)

Updates the value of the input tensor through the addition operation.

Using given values to update tensor value through the add operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each *i, ···, j* in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] \mathrel{+}= \text{updates}[i, ..., j, :]$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Note: This is an in-place update operator. Therefore, the *input_x* will be updated after the operation is completed.

Parameters

use_locking (*bool, optional*) −Whether to protect the assignment by a lock. If `True` , *input_x* will be protected by the lock. Otherwise, the calculation result is undefined. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor.
- **indices** (Tensor) - The index to do min operation whose data type must be mindspore.int32 or mindspore.int64.
- **updates** (Tensor) - The tensor doing the min operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape* + *x.shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** −If *use_locking* is not a bool.
- **TypeError** −If *indices* is not an int32 or an int64.
- **ValueError** −If the shape of *updates* is not equal to *indices.shape* + *x.shape[1:]*.
- **RuntimeError** −If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
↪float32), name="x")
>>> indices = Tensor(np.array([[0, 1], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.ones([2, 2, 3]), mindspore.float32)
>>> scatter_add = ops.ScatterAdd()
>>> output = scatter_add(input_x, indices, updates)
>>> print(output)
[[1. 1. 1.]
 [3. 3. 3.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
↪initialized.
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
↪float32), name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [0.0, 0.0, 0.0] + [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] + [3.0, 3.0, 3.0] = [3.0, 3.0, 3.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [3.0, 3.0, 3.0] + [7.0, 7.0, 7.0] = [10.0, 10.0, 10.0]
>>> # input_x[1] = [10.0, 10.0, 10.0] + [9.0, 9.0, 9.0] = [19.0, 19.0, 19.0]
>>> indices = Tensor(np.array([[0, 1], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_add = ops.ScatterAdd()
>>> output = scatter_add(input_x, indices, updates)
>>> print(output)
[[ 1.  1.  1.]
 [19. 19. 19.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
↪initialized.
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
↪float32), name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [0.0, 0.0, 0.0] + [3.0, 3.0, 3.0] = [3.0, 3.0, 3.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] + [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [1.0, 1.0, 1.0] + [7.0, 7.0, 7.0] = [8.0, 8.0, 8.0]
>>> # input_x[1] = [8.0, 8.0, 8.0] + [9.0, 9.0, 9.0] = [17.0, 17.0, 17.0]
>>> indices = Tensor(np.array([[1, 0], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_add = ops.ScatterAdd()
>>> output = scatter_add(input_x, indices, updates)
>>> print(output)
[[ 3.  3.  3.]
 [17. 17. 17.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
↪initialized.
```

(continues on next page)

(continued from previous page)

```
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
↪float32), name="x")
>>> # for indices = [[0, 1], [0, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [0.0, 0.0, 0.0] + [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] + [3.0, 3.0, 3.0] = [3.0, 3.0, 3.0]
>>> # step 2: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] + [7.0, 7.0, 7.0] = [8.0, 8.0, 8.0]
>>> # input_x[1] = [3.0, 3.0, 3.0] + [9.0, 9.0, 9.0] = [12.0, 12.0, 12.0]
>>> indices = Tensor(np.array([[0, 1], [0, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_add = ops.ScatterAdd()
>>> output = scatter_add(input_x, indices, updates)
>>> print(output)
[[ 8.  8.  8.]
 [12. 12. 12.]]
```

18.5.5 mindspore.ops.ScatterDiv

class mindspore.ops.**ScatterDiv**(*use_locking=False*)

Updates the value of the input tensor through the divide operation.

Using given values to update tensor value through the div operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each $i, ..., j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] \ /= \text{updates}[i, ..., j, :]$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type. A RuntimeError will be reported when *updates* does not support conversion to the data type required by *input_x*.

Parameters

use_locking (*bool*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do divide operation whose data type must be mstype.int32 or mstype.int64.
- **updates** (Tensor) - The tensor doing the divide operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape* + *input_x.shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *indices* is not an int32 or an int64.
- **ValueError** – If the shape of *updates* is not equal to *indices.shape* + *input_x.shape[1:]*.

- **RuntimeError** –If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.
- **RuntimeError** –On the Ascend platform, the input data dimension of *input_x* , *indices* and *updates* is greater than 8 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[6.0, 6.0, 6.0], [2.0, 2.0, 2.0]]), mstype.float32),
 ↪ name="x")
>>> indices = Tensor(np.array([0, 1]), mstype.int32)
>>> updates = Tensor(np.array([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]), mstype.float32)
>>> scatter_div = ops.ScatterDiv()
>>> output = scatter_div(input_x, indices, updates)
>>> print(output)
[[3. 3. 3.]
 [1. 1. 1.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[105.0, 105.0, 105.0],
...                                      [315.0, 315.0, 315.0]]), mstype.float32), name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [1.0, 1.0, 1.0] = [105.0, 105.0, 105.0]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [3.0, 3.0, 3.0] = [105.0, 105.0, 105.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [105.0, 105.0, 105.0] / [5.0, 5.0, 5.0] = [21.0, 21.0, 21.0]
>>> # input_x[1] = [21.0, 21.0, 21.0] / [7.0, 7.0, 7.0] = [3.0, 3.0, 3.0]
>>> indices = Tensor(np.array([[0, 1], [1, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[5.0, 5.0, 5.0], [7.0, 7.0, 7.0]]]), mstype.float32)
>>> scatter_div = ops.ScatterDiv()
>>> output = scatter_div(input_x, indices, updates)
>>> print(output)
[[105. 105. 105.]
 [  3.   3.   3.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[105.0, 105.0, 105.0],
...                                      [315.0, 315.0, 315.0]]), mstype.float32), name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [3.0, 3.0, 3.0] = [35.0, 35.0, 35.0]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [1.0, 1.0, 1.0] = [315.0, 315.0, 315.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [5.0, 5.0, 5.0] = [63.0 63.0 63.0]
>>> # input_x[1] = [63.0 63.0 63.0] / [7.0, 7.0, 7.0] = [9.0, 9.0, 9.0]
>>> indices = Tensor(np.array([[1, 0], [1, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[5.0, 5.0, 5.0], [7.0, 7.0, 7.0]]]), mstype.float32)
```

(continues on next page)

(continued from previous page)

```
>>> scatter_div = ops.ScatterDiv()
>>> output = scatter_div(input_x, indices, updates)
>>> print(output)
[[35. 35. 35.]
 [ 9.  9.  9.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[105.0, 105.0, 105.0],
...                                      [315.0, 315.0, 315.0]]), mstype.float32), name="x")
>>> # for indices = [[0, 1], [0, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [1.0, 1.0, 1.0] = [105.0, 105.0, 105.0]
>>> # input_x[1] = [315.0, 315.0, 315.0] / [3.0, 3.0, 3.0] = [105.0, 105.0, 105.0]
>>> # step 2: [0, 1]
>>> # input_x[0] = [105.0, 105.0, 105.0] / [5.0, 5.0, 5.0] = [21.0, 21.0, 21.0]
>>> # input_x[1] = [105.0, 105.0, 105.0] / [7.0, 7.0, 7.0] = [15.0, 15.0, 15.0]
>>> indices = Tensor(np.array([[0, 1], [0, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[5.0, 5.0, 5.0], [7.0, 7.0, 7.0]]]), mstype.float32)
>>> scatter_div = ops.ScatterDiv()
>>> output = scatter_div(input_x, indices, updates)
>>> print(output)
[[21. 21. 21.]
 [15. 15. 15.]]
```

18.5.6 mindspore.ops.ScatterMax

class mindspore.ops.**ScatterMax**(*use_locking=False*)

Updates the value of the input tensor through the maximum operation.

Using given values to update tensor value through the max operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each $i, ..., j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \max(\text{input\_x}[\text{indices}[i, ..., j], :], \text{updates}[i, ..., j, :])$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type. A RuntimeError will be reported when *updates* does not support conversion to the data type required by *input_x*.

Parameters

use_locking (*bool*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do max operation whose data type must be mindspore.int32 or mindspore.int64.
- **updates** (Tensor) - The tensor that performs the maximum operation with *input_x*, the data type is the same as *input_x*, the shape is *indices.shape* + *input_x.shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** –If *use_locking* is not a bool.
- **TypeError** –If *indices* is not an int32 or an int64.
- **ValueError** –If the shape of *updates* is not equal to *indices.shape* + *x.shape[1:]*.
- **RuntimeError** –If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.
- **RuntimeError** –On the Ascend platform, the input data dimension of *input_x* , *indices* and *updates* is greater than 8 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]), mindspore.
↪float32),
...                     name="input_x")
>>> indices = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.ones([2, 2, 3]) * 88, mindspore.float32)
>>> scatter_max = ops.ScatterMax()
>>> output = scatter_max(input_x, indices, updates)
>>> print(output)
[[88. 88. 88.]
 [88. 88. 88.]]
```

18.5.7 mindspore.ops.ScatterMin

class mindspore.ops.**ScatterMin**(*use_locking=False*)

Updates the value of the input tensor through the minimum operation.

Using given values to update tensor value through the min operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each $i, ..., j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \min(\text{input\_x}[\text{indices}[i, ..., j], :], \text{updates}[i, ..., j, :])$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type. A RuntimeError will be reported when *updates* does not support conversion to the data type required by *input_x*.

Parameters

use_locking (*bool, optional*) –Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.

- **indices** (Tensor) - The index to do min operation whose data type must be mindspore.int32 or mindspore.int64.
- **updates** (Tensor) - The tensor doing the min operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape + input_x.shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *indices* is not an int32 or an int64.
- **ValueError** – If the shape of *updates* is not equal to *indices.shape + input_x.shape[1:]*.
- **RuntimeError** – If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.
- **RuntimeError** – On the Ascend platform, the input data dimension of *input_x* , *indices* and *updates* is greater than 8 dimensions.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[0.0, 1.0, 2.0], [0.0, 0.0, 0.0]]), mindspore.
↪float32),
...                     name="input_x")
>>> indices = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> update = Tensor(np.ones([2, 2, 3]), mindspore.float32)
>>> scatter_min = ops.ScatterMin()
>>> output = scatter_min(input_x, indices, update)
>>> print(output)
[[0. 1. 1.]
 [0. 0. 0.]]
```

18.5.8 mindspore.ops.ScatterMul

class mindspore.ops.**ScatterMul**(*use_locking=False*)

Updates the value of the input tensor through the multiply operation.

Using given values to update tensor value through the mul operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each $i, \cdots, j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] \mathrel{*}= \text{updates}[i, ..., j, :]$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool*) −Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do multiply operation whose data type must be mstype.int32 or mstype.int64.
- **updates** (Tensor) - The tensor doing the multiply operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape + input_x.shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** −If *use_locking* is not a bool.
- **TypeError** −If *indices* is not an int32 or an int64.
- **ValueError** −If the shape of *updates* is not equal to *indices.shape + input_x.shape[1:]*.
- **RuntimeError** −If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]), mstype.float32),
 ↪ name="x")
>>> indices = Tensor(np.array([0, 1]), mstype.int32)
>>> updates = Tensor(np.array([[2.0, 2.0, 2.0], [2.0, 2.0, 2.0]]), mstype.float32)
>>> scatter_mul = ops.ScatterMul()
>>> output = scatter_mul(input_x, indices, updates)
>>> print(output)
[[2. 2. 2.]
 [4. 4. 4.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]), mstype.float32),
 ↪ name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [3.0, 3.0, 3.0] = [6.0, 6.0, 6.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [6.0, 6.0, 6.0] * [7.0, 7.0, 7.0] = [42.0, 42.0, 42.0]
>>> # input_x[1] = [42.0, 42.0, 42.0] * [9.0, 9.0, 9.0] = [378.0, 378.0, 378.0]
>>> indices = Tensor(np.array([[0, 1], [1, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mstype.float32)
```

(continues on next page)

(continued from previous page)

```
>>> scatter_mul = ops.ScatterMul()
>>> output = scatter_mul(input_x, indices, updates)
>>> print(output)
[[  1.   1.   1.]
 [378. 378. 378.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]), mstype.float32),
 ↪ name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [3.0, 3.0, 3.0] = [3.0, 3.0, 3.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [1.0, 1.0, 1.0] = [2.0, 2.0, 2.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [7.0, 7.0, 7.0] = [14.0, 14.0, 14.0]
>>> # input_x[1] = [14.0, 14.0, 14.0] * [9.0, 9.0, 9.0] = [126.0, 126.0, 126.0]
>>> indices = Tensor(np.array([[1, 0], [1, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mstype.float32)
>>> scatter_mul = ops.ScatterMul()
>>> output = scatter_mul(input_x, indices, updates)
>>> print(output)
[[  3.   3.   3.]
 [126. 126. 126.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]), mstype.float32),
 ↪ name="x")
>>> # for indices = [[0, 1], [0, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [1.0, 1.0, 1.0] = [1.0, 1.0, 1.0]
>>> # input_x[1] = [2.0, 2.0, 2.0] * [3.0, 3.0, 3.0] = [6.0, 6.0, 6.0]
>>> # step 2: [0, 1]
>>> # input_x[0] = [1.0, 1.0, 1.0] * [7.0, 7.0, 7.0] = [7.0, 7.0, 7.0]
>>> # input_x[1] = [6.0, 6.0, 6.0] * [9.0, 9.0, 9.0] = [54.0, 54.0, 54.0]
>>> indices = Tensor(np.array([[0, 1], [0, 1]]), mstype.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mstype.float32)
>>> scatter_mul = ops.ScatterMul()
>>> output = scatter_mul(input_x, indices, updates)
>>> print(output)
[[ 7.  7.  7.]
 [54. 54. 54.]]
```

18.5.9 mindspore.ops.ScatterNdAdd

class mindspore.ops.**ScatterNdAdd**(*use_locking=False*)

Applies sparse addition to individual values or slices in a tensor.

Using given values to update tensor value through the add operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

Refer to *mindspore.ops.scatter_nd_add()* for more details.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do add operation whose data type must be mindspore.int32. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor doing the add operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8]), mindspore.float32), name=
 ↪"x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> use_locking = False
>>> scatter_nd_add = ops.ScatterNdAdd(use_locking)
>>> output = scatter_nd_add(input_x, indices, updates)
>>> print(output)
[ 1. 10.  9.  4. 12.  6.  7. 17.]
>>> input_x = Parameter(Tensor(np.zeros((4, 4, 4)), mindspore.int32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
 ↪mindspore.int32)
>>> use_locking = False
>>> scatter_nd_add = ops.ScatterNdAdd(use_locking)
>>> output = scatter_nd_add(input_x, indices, updates)
>>> print(output)
[[[1 1 1 1]
  [2 2 2 2]
  [3 3 3 3]
  [4 4 4 4]]
 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]
 [[5 5 5 5]
  [6 6 6 6]
  [7 7 7 7]
  [8 8 8 8]]
 [[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]]
```

18.5.10 mindspore.ops.ScatterNdSub

class mindspore.ops.**ScatterNdSub**(*use_locking=False*)

Applies sparse subtraction to individual values or slices in a tensor.

Using given values to update tensor value through the subtraction operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

Refer to *mindspore.ops.scatter_nd_sub()* for more details.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do sub operation whose data type must be mindspore.int32. The rank of indices must be at least 2 and *indices.shape[-1] <= len(shape)*.
- **updates** (Tensor) - The tensor doing the sub operation with *input_x*, the data type is same as *input_x*, the shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([1, 2, 3, 4, 5, 6, 7, 8]), mindspore.float32), name=
 ↪"x")
>>> indices = Tensor(np.array([[2], [4], [1], [7]]), mindspore.int32)
>>> updates = Tensor(np.array([6, 7, 8, 9]), mindspore.float32)
>>> use_locking = False
>>> scatter_nd_sub = ops.ScatterNdSub(use_locking)
>>> output = scatter_nd_sub(input_x, indices, updates)
>>> print(output)
[ 1. -6. -3.  4. -2.  6.  7. -1.]
>>> input_x = Parameter(Tensor(np.zeros((4, 4, 4)), mindspore.int32))
>>> indices = Tensor(np.array([[0], [2]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3], [4, 4, 4, 4]],
...                            [[5, 5, 5, 5], [6, 6, 6, 6], [7, 7, 7, 7], [8, 8, 8, 8]]]),␣
 ↪mindspore.int32)
>>> use_locking = False
>>> scatter_nd_sub = ops.ScatterNdSub(use_locking)
>>> output = scatter_nd_sub(input_x, indices, updates)
>>> print(output)
[[[-1 -1 -1 -1]
  [-2 -2 -2 -2]
  [-3 -3 -3 -3]
```

(continues on next page)

(continued from previous page)

```
  [-4 -4 -4 -4]]
 [[ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]]
 [[-5 -5 -5 -5]
  [-6 -6 -6 -6]
  [-7 -7 -7 -7]
  [-8 -8 -8 -8]]
 [[ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]
  [ 0  0  0  0]]]
```

18.5.11 mindspore.ops.ScatterNdUpdate

class mindspore.ops.**ScatterNdUpdate**(*use_locking=True*)

Updates tensor values by using input indices and value.

Using given values to update tensor value, along with the input indices.

input_x has rank P and *indices* has rank Q where $Q >= 2$.

indices has shape $(i_0, i_1, ..., i_{Q-2}, N)$ where $N <= P$.

The last dimension of *indices* (with length N) indicates slices along the N th dimension of *input_x*.

updates is a tensor of rank $Q-1+P-N$, and its shape is: $(i_0, i_1, ..., i_{Q-2}, x\_shape_N, ..., x\_shape_{P-1})$.

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `True` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index of input tensor, with int32 or int64 data type.
- **updates** (Tensor) - N-D(2D or 3D) Tensor The tensor to be updated to the input tensor, has the same type as input. The shape is *indices.shape[:-1] + x.shape[indices.shape[-1]:]*.

Outputs:

Tensor, has the same shape and type as *input_x*.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *indices* is not an int32 or an int64.
- **RuntimeError** – If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> np_x = np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]])
>>> input_x = mindspore.Parameter(Tensor(np_x, mindspore.float32), name="x")
>>> indices = Tensor(np.array([[0, 0], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([1.0, 2.2]), mindspore.float32)
>>> op = ops.ScatterNdUpdate()
>>> output = op(input_x, indices, updates)
>>> print(output)
[[1.   0.3   3.6]
 [0.4  2.2  -3.2]]
```

18.5.12 mindspore.ops.ScatterNonAliasingAdd

class mindspore.ops.**ScatterNonAliasingAdd**

Please use TensorScatterAdd instead.

Supported Platforms:

Deprecated

18.5.13 mindspore.ops.ScatterSub

class mindspore.ops.**ScatterSub**(*use_locking=False*)

Updates the value of the input tensor through the subtraction operation.

Using given values to update tensor value through the subtraction operation, along with the input indices. This operation outputs the *input_x* after the update is done, which makes it convenient to use the updated value.

for each $i, \cdots, j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] \mathrel{-}= \text{updates}[i, ..., j, :]$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool, optional*) – Whether to protect the assignment by a lock. Default: `False` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index to do min operation whose data type must be mindspore.int32 or mindspore.int64.
- **updates** (Tensor) - The tensor doing the min operation with *input_x*, the data type is same as *input_x*, the shape is *indices_shape* + *x_shape[1:]*.

Outputs:

Tensor, the updated *input_x*, has the same shape and type as *input_x*.

Raises

- **TypeError** – If *use_locking* is not a bool.
- **TypeError** – If *indices* is not an int32.
- **ValueError** – If the shape of *updates* is not equal to *indices_shape* + *x_shape[1:]*.
- **RuntimeError** – If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops, Parameter
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [1.0, 1.0, 1.0]]), mindspore.
 ↪float32), name="x")
>>> indices = Tensor(np.array([[0, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [2.0, 2.0, 2.0]]]), mindspore.float32)
>>> scatter_sub = ops.ScatterSub()
>>> output = scatter_sub(input_x, indices, updates)
>>> print(output)
[[-1. -1. -1.]
 [-1. -1. -1.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
 ↪float32), name="x")
>>> # for indices = [[0, 1], [1, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [0.0, 0.0, 0.0] - [1.0, 1.0, 1.0] = [-1.0, -1.0, -1.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] - [3.0, 3.0, 3.0] = [-3.0, -3.0, -3.0]
>>> # step 2: [1, 1]
>>> # input_x[1] = [-3.0, -3.0, -3.0] - [7.0, 7.0, 7.0] = [-10.0, -10.0, -10.0]
>>> # input_x[1] = [-10.0, -10.0, -10.0] - [9.0, 9.0, 9.0] = [-19.0, -19.0, -19.0]
>>> indices = Tensor(np.array([[0, 1], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_sub = ops.ScatterSub()
>>> output = scatter_sub(input_x, indices, updates)
>>> print(output)
[[ -1.  -1.  -1.]
 [-19. -19. -19.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
 ↪float32), name="x")
>>> # for indices = [[1, 0], [1, 1]]
>>> # step 1: [1, 0]
>>> # input_x[0] = [0.0, 0.0, 0.0] - [3.0, 3.0, 3.0] = [-3.0, -3.0, -3.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] - [1.0, 1.0, 1.0] = [-1.0, -1.0, -1.0]
```

(continues on next page)

(continued from previous page)

```
>>> # step 2: [1, 1]
>>> # input_x[1] = [-1.0, -1.0, -1.0] - [7.0, 7.0, 7.0] = [-8.0, -8.0, -8.0]
>>> # input_x[1] = [-8.0, -8.0, -8.0] - [9.0, 9.0, 9.0] = [-17.0, -17.0, -17.0]
>>> indices = Tensor(np.array([[1, 0], [1, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_sub = ops.ScatterSub()
>>> output = scatter_sub(input_x, indices, updates)
>>> print(output)
[[ -3.  -3.  -3.]
 [-17. -17. -17.]]
>>> # for input_x will be updated after the operation is completed. input_x need to be re-
 ↪initialized.
>>> input_x = Parameter(Tensor(np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]), mindspore.
 ↪float32), name="x")
>>> # for indices = [[0, 1], [0, 1]]
>>> # step 1: [0, 1]
>>> # input_x[0] = [0.0, 0.0, 0.0] - [1.0, 1.0, 1.0] = [-1.0, -1.0, -1.0]
>>> # input_x[1] = [0.0, 0.0, 0.0] - [3.0, 3.0, 3.0] = [-3.0, -3.0, -3.0]
>>> # step 2: [0, 1]
>>> # input_x[0] = [-1.0, -1.0, -1.0] - [7.0, 7.0, 7.0] = [-8.0, -8.0, -8.0]
>>> # input_x[1] = [-3.0, -3.0, -3.0] - [9.0, 9.0, 9.0] = [-12.0, -12.0, -12.0]
>>> indices = Tensor(np.array([[0, 1], [0, 1]]), mindspore.int32)
>>> updates = Tensor(np.array([[[1.0, 1.0, 1.0], [3.0, 3.0, 3.0]],
...                            [[7.0, 7.0, 7.0], [9.0, 9.0, 9.0]]]), mindspore.float32)
>>> scatter_sub = ops.ScatterSub()
>>> output = scatter_sub(input_x, indices, updates)
>>> print(output)
[[ -8.  -8.  -8.]
 [-12. -12. -12.]]
```

18.5.14 mindspore.ops.ScatterUpdate

class mindspore.ops.**ScatterUpdate**(*use_locking=True*)

Updates tensor values by using input indices and value.

Using given values to update tensor value, along with the input indices.

for each $i, \cdots, j$ in *indices.shape*:

$$\text{input\_x}[\text{indices}[i, ..., j], :] = \text{updates}[i, ..., j, :]$$

Inputs of *input_x* and *updates* comply with the implicit type conversion rules to make the data types consistent. If they have different data types, the lower priority data type will be converted to the relatively highest priority data type.

Parameters

use_locking (*bool*) – Whether to protect the assignment by a lock. Default: `True` .

Inputs:

- **input_x** (Union[Parameter, Tensor]) - The target tensor, with data type of Parameter or Tensor. The shape is 0-D or $(N, *)$ where $*$ means any number of additional dimensions.
- **indices** (Tensor) - The index of input tensor. With int32 data type. If there are duplicates in indices, the order for updating is undefined.

- **updates** (Tensor) - The tensor to update the input tensor, has the same type as input, and updates.shape = indices.shape + input_x.shape[1:].

Outputs:

Tensor, has the same shape and type as *input_x*.

Raises

- **TypeError** –If *use_locking* is not a bool.
- **TypeError** –If *indices* is not an int32.
- **ValueError** –If the shape of *updates* is not equal to *indices.shape* + *input_x.shape[1:]*.
- **RuntimeError** –If the data type of *input_x* and *updates* conversion is required when data type conversion is not supported.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> np_x = np.array([[-0.1, 0.3, 3.6], [0.4, 0.5, -3.2]])
>>> input_x = mindspore.Parameter(Tensor(np_x, mindspore.float32), name="x")
>>> indices = Tensor(np.array([0, 1]), mindspore.int32)
>>> np_updates = np.array([[2.0, 1.2, 1.0], [3.0, 1.2, 1.0]])
>>> updates = Tensor(np_updates, mindspore.float32)
>>> op = ops.ScatterUpdate()
>>> output = op(input_x, indices, updates)
>>> print(output)
[[2. 1.2  1.]
 [3. 1.2  1.]]
```

18.6 Data Operation Operator

API Name	Description	Supported Platforms
mindspore.ops.GetNext	Returns the next element in the dataset queue.	`Ascend` `GPU`

18.6.1 mindspore.ops.GetNext

class mindspore.ops.**GetNext**(*types*, *shapes*, *output_num*, *shared_name*)

Returns the next element in the dataset queue.

Note: The GetNext operation needs to be associated with network and it also depends on the 'dataset' interface, For example, please refer to `mindspore.dataset.MnistDataset` . it can't be used directly as a single operation. For details, please refer to *mindspore.connect _network_with_dataset* source code.

Parameters

- **types** (list[*mindspore.dtype*]) –The type of the outputs.
- **shapes** (*list[tuple[int]]*) –The dimensionality of the outputs.
- **output_num** (*int*) –The output number, length of *types* and *shapes*.
- **shared_name** (*str*) –Queue name to fetch the data.

Inputs:

No inputs.

Outputs:

tuple[Tensor], the output of dataset. The shape is described in *shapes* and the type is described in *types*.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> from mindspore import ops
>>> from mindspore import dataset as ds
>>> from mindspore import dtype as mstype
>>> data_path = "/path/to/MNIST_Data/train/"
>>> train_dataset = ds.MnistDataset(data_path, num_samples=10)
>>> dataset_helper = mindspore.DatasetHelper(train_dataset, dataset_sink_mode=True)
>>> dataset = dataset_helper.iter.dataset
>>> dataset_types, dataset_shapes = dataset_helper.types_shapes()
>>> queue_name = dataset.__transfer_dataset__.queue_name
>>> get_next = ops.GetNext(dataset_types, dataset_shapes, len(dataset_types), queue_name)
>>> data, label = get_next()
>>> relu = ops.ReLU()
>>> result = relu(data.astype(mstype.float32))
>>> print(result.shape)
(28, 28, 1)
```

18.7 Communication Operator

Distributed training involves communication operations for data transfer. For more details, refer to Distributed Set Communication Primitives .

Note that the APIs in the following list need to preset communication environment variables. For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

API Name	Description	Supported Platforms
mindspore.ops.AllGather	Gathers tensors from the specified communication group and returns the tensor which is all gathered.	`Ascend` `GPU`
mindspore.ops.AllReduce	Reduces tensors across all devices in such a way that all devices will get the same final result, returns the tensor which is all reduced.	`Ascend` `GPU` `CPU`

continues on next page

Table 21 – continued from previous page

mindspore.ops.AlltoAll	AlltoAll is a collective operation.	Ascend
mindspore.ops.AlltoAllV	AllToAllV which support uneven scatter and gather compared with AllToAll.	Ascend
mindspore.ops.Barrier	Synchronizes all processes in the specified group.	Ascend
mindspore.ops.Broadcast	Broadcasts the tensor to the whole group.	Ascend GPU
mindspore.ops.CollectiveGather	Gathers tensors from the specified communication group.	Ascend
mindspore.ops.CollectiveScatter	Scatter input data evently across the processes in the specified communication group.	Ascend
mindspore.ops.NeighborExchangeV2	NeighborExchangeV2 is a collective communication operation.	Ascend
mindspore.ops.Receive	Receive tensors from src_rank.	Ascend GPU
mindspore.ops.ReduceOp	Operation options for reducing tensors.	Ascend GPU
mindspore.ops.ReduceScatter	Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.	Ascend GPU
mindspore.ops.Reduce	Reduces tensors across the processes in the specified communication group, sends the result to the target dest_rank(local rank), and returns the tensor which is sent to the target process.	Ascend
mindspore.ops.Send	Send tensors to the specified dest_rank.	Ascend GPU

18.7.1 mindspore.ops.AllGather

class mindspore.ops.**AllGather**(*group=GlobalComm.WORLD_COMM_GROUP*)

Gathers tensors from the specified communication group and returns the tensor which is all gathered.

Note:

- The tensors must have the same shape and format in all processes of the collection.

Parameters

group (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor. If the number of devices in the group is N, then the shape of output is $(N, x_1, x_2, ..., x_R)$.

Raises

- **TypeError** – If *group* is not a str.
- **ValueError** – If the local rank id of the calling process in the group is larger than the group's rank size.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import ops
>>> import mindspore.nn as nn
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.allgather = ops.AllGather()
...
...     def construct(self, x):
...         return self.allgather(x)
...
>>> input_x = Tensor(np.ones([2, 8]).astype(np.float32))
>>> net = Net()
>>> output = net(input_x)
>>> print(output)
[[1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1.]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - AllGather

18.7.2 mindspore.ops.AllReduce

class mindspore.ops.**AllReduce**(*op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*)

Reduces tensors across all devices in such a way that all devices will get the same final result, returns the tensor which is all reduced.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **op** (*str, optional*) – Specifies an operation used for element-wise reductions, like sum, prod, max, and min. On the CPU, only 'sum' is supported. Default: `ReduceOp.SUM`.

- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP` , which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor, has the same shape of the input, i.e., $(x_1, x_2, ..., x_R)$. The contents depend on the specified operation.

Raises

- **TypeError** –If any of *op* and *group* is not a str or the input's dtype is bool.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>> from mindspore.ops import ReduceOp
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>>
>>> init()
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.allreduce_sum = ops.AllReduce(ReduceOp.SUM)
...
...     def construct(self, x):
...         return self.allreduce_sum(x)
...
>>> input_ = Tensor(np.ones([2, 8]).astype(np.float32))
>>> net = Net()
>>> output = net(input_)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - AllReduce

18.7.3 mindspore.ops.AlltoAll

class mindspore.ops.**AlltoAll**(*split_count*, *split_dim*, *concat_dim*, *group=GlobalComm.WORLD_COMM_GROUP*)

AlltoAll is a collective operation.

AlltoAll sends data from the all processes to the all processes in the specified group. It has two phases:

- The scatter phase: On each process, the operand is split into split_count number of blocks along the split_dimensions, and the blocks are scattered to all processes, e.g., the ith block is send to the ith process.
- The gather phase: Each process concatenates the received blocks along the concat_dimension.

Note: This operator requires a full-mesh network topology, each device has the same vlan id, and the ip & mask are in the same subnet, please check the details .

Parameters

- **split_count** (*int*) –On each process, divide blocks into split_count number.
- **split_dim** (*int*) –On each process, split blocks along the split_dim.
- **concat_dim** (*int*) –On each process, gather the received blocks along the concat_dimension.
- **group** (*str, optional*) –The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP .

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor. If the shape of input tensor is $(x_1, x_2, ..., x_R)$, then the shape of output tensor is $(y_1, y_2, ..., y_R)$, where:

- $y_{split\_dim} = x_{split\_dim}/split\_count$
- $y_{concat\_dim} = x_{concat\_dim} * split\_count$
- $y_{other} = x_{other}$.

Raises

TypeError –If group is not a string.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 8 devices.

```
>>> import os
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> import numpy as np
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.alltoall = ops.AlltoAll(split_count = 8, split_dim = -2, concat_dim = -1)
...
...     def construct(self, x):
...         out = self.alltoall(x)
...         return out
...
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> net = Net()
>>> rank_id = int(os.getenv("RANK_ID"))
>>> input_x = Tensor(np.ones([1, 1, 8, 1]) * rank_id, dtype = ms.float32)
>>> output = net(input_x)
>>> print(output)
[[[[0. 1. 2. 3. 4. 5. 6. 7.]]]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - AlltoAll

18.7.4 mindspore.ops.AlltoAllV

class mindspore.ops.**AlltoAllV**(*group=GlobalComm.WORLD_COMM_GROUP*, *block_size=1*)

AllToAllV which support uneven scatter and gather compared with AllToAll.

Note:

- Only support flatten tensor as input. input tensor should be flattened and concatenated before call this primitive.

Parameters

- **group** (*str*, *optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend.
- **block_size** (*int*, *optional*) – The basic units for scatter and gather numel by *send_numel_list* and *recv_numel_list*. Default: `1`.

Inputs:

- **input_x** (Tensor) - flatten tensor to scatter. The shape of tensor is (x_1).
- **send_numel_list** (Union[tuple[int], list[int], Tensor]) - split numel to scatter to different remote rank. The actual distributed data numel is $(send_numel_list * block_size * input_x.dtype)$.
- **recv_numel_list** (Union[tuple[int], list[int], Tensor]) - split numel to gather from different remote rank. The actual aggregated data numel is $(recv_numel_list * block_size * input_x.dtype)$.

Outputs:

Tensor, flattened and concatenated tensor gather from remote ranks. If gather result is empty, it will return a Tensor with shape *()*, and value has no actual meaning.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies.

Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> from mindspore import ops
>>> import mindspore.nn as nn
>>> from mindspore.communication import init, get_rank
>>> from mindspore import Tensor
>>>
>>> init()
>>> rank = get_rank()
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.all_to_all = ops.AlltoAllV()
...
...     def construct(self, x, send_numel_list, recv_numel_list):
...         return self.all_to_all(x, send_numel_list, recv_numel_list)
>>> send_numel_list = []
>>> recv_numel_list = []
>>> if rank == 0:
...     send_tensor = Tensor([0, 1, 2.])
...     send_numel_list = [1, 2]
...     recv_numel_list = [1, 2]
>>> elif rank == 1:
...     send_tensor = Tensor([3, 4, 5.])
...     send_numel_list = [2, 1]
...     recv_numel_list = [2, 1]
>>> net = Net()
>>> output = net(send_tensor, send_numel_list, recv_numel_list)
>>> print(output)
rank 0:
[0. 3. 4]
rank 1:
[1. 2. 5]
```

18.7.5 mindspore.ops.Barrier

class mindspore.ops.**Barrier**(*group=GlobalComm.WORLD_COMM_GROUP*)

Synchronizes all processes in the specified group. Once the process call this operation, it will be blocked until all processes call this operation. After all processes finish calling the operations, the blocked processes will be waken and continue their task.

Parameters

group (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.

Raises

- **TypeError** – If *group* is not a str.
- **RuntimeError** – If backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> # Launch 4 processes.
>>> init()
>>> class BarrierNet(nn.Cell):
...     def __init__(self):
...         super(BarrierNet, self).__init__()
...         self.barrier = ops.Barrier()
...
...     def construct(self):
...         self.barrier()
>>> net = BarrierNet()
>>> net()
```

Tutorial Examples:

- Distributed Set Communication Primitives - Barrier

18.7.6 mindspore.ops.Broadcast

class mindspore.ops.**Broadcast**(*root_rank*, *group=GlobalComm.WORLD_COMM_GROUP*)

Broadcasts the tensor to the whole group.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **root_rank** (*int*) –Specifies the rank(global rank) of the process that broadcast the tensor. And only process *root_rank* will broadcast the tensor.
- **group** (*str*, *optional*) –The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP .

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

tuple[Tensor], Tensor has the same shape of the input, i.e., $(x_1, x_2, ..., x_R)$. The contents depend on the data of the *root_rank* device.

Raises

TypeError –If root_rank is not an integer or group is not a string.

Supported Platforms:

Ascend GPU

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> import numpy as np
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.broadcast = ops.Broadcast(1)
```

(continues on next page)

(continued from previous page)

```
...
...     def construct(self, x):
...         return self.broadcast((x,))
...
>>> input_x = Tensor(np.ones([2, 4]).astype(np.int32))
>>> net = Net()
>>> output = net(input_x)
>>> print(output)
(Tensor(shape[2,4], dtype=Int32, value=
[[1, 1, 1, 1],
 [1, 1, 1, 1]]),)
```

Tutorial Examples:

- Distributed Set Communication Primitives - Broadcast

18.7.7 mindspore.ops.CollectiveGather

class mindspore.ops.**CollectiveGather**(*dest_rank*, *group=GlobalComm.WORLD_COMM_GROUP*)

Gathers tensors from the specified communication group. The operation will gather the tensor from processes according to dimension 0.

Note: Only the tensor in process *dest_rank* (global rank) will keep the gathered tensor. The other process will keep a tensor with shape [1], which has no mathematical meaning.

Parameters

- **dest_rank** (*int*) –Specifies the rank of the process that receive the tensor. And only process *dest_rank* will receive the gathered tensor.
- **group** (*str, optional*) –The communication group to work on. Default: GlobalComm.WORLD_COMM_GROUP.

Inputs:

- **input_x** (Tensor) - The tensor to be gathered. The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor, the shape of output is $(\sum x_1, x_2, ..., x_R)$. The dimension 0 of data is equal to sum of the dimension of input tensor, and the other dimension keep the same.

Raises

- **TypeError** –If *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.
- **ValueError** –If the local rank id of the calling process in the group is larger than the group's rank size.

Supported Platforms:

Ascend

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> import numpy as np
>>> import mindspore as ms
>>> import mindspore.nn as nn
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> # Launch 2 processes.
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> class CollectiveGatherNet(nn.Cell):
...     def __init__(self):
...         super(CollectiveGatherNet, self).__init__()
...         self.collective_gather = ops.CollectiveGather(dest_rank=0)
...
...     def construct(self, x):
...         return self.collective_gather(x)
...
>>> input = Tensor(np.arange(4).reshape([2, 2]).astype(np.float32))
>>> net = CollectiveGatherNet()
>>> output = net(input)
>>> print(output)
Process with rank 0: [[0. 1.],
                      [2. 3.],
                      [0. 1.],
                      [2. 3.]]
Process with rank 1: [0.]
```

Tutorial Examples:

- Distributed Set Communication Primitives - CollectiveGather

18.7.8 mindspore.ops.CollectiveScatter

class mindspore.ops.**CollectiveScatter**(*src_rank=0*, *group=GlobalComm.WORLD_COMM_GROUP*)

Scatter input data evently across the processes in the specified communication group.

Note: The interface behavior only support Tensor input and scatter evenly. Only the tensor in process *src_rank* (global rank) will do scatter.

Parameters

- **src_rank** (*int, optional*) – Specifies the rank of the process that send the tensor. And only process *src_rank* will send the tensor.

- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.

Inputs:

- **input_x** (Tensor) - The input tensor to be scattered. The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor, the shape of output is $(x_1/src\_rank, x_2, ..., x_R)$. The dimension 0 of data is equal to the dimension of input tensor divided by *src*, and the other dimension keep the same.

Raises

- **TypeError** –If *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.
- **ValueError** –If the local rank id of the calling process in the group is larger than the group's rank size.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> from mindspore.communication.management import init, get_rank
>>> from mindspore import ops
>>> # Launch 2 processes.
>>> init()
>>> class CollectiveScatterNet(nn.Cell):
...     def __init__(self):
...         super(CollectiveScatterNet, self).__init__()
...         self.collective_scatter = ops.CollectiveScatter(src_rank=0)
...
...     def construct(self, x):
...         return self.collective_scatter(x)
...
>>> input = Tensor(np.arange(8).reshape([4, 2]).astype(np.float32))
>>> net = CollectiveScatterNet()
>>> output = net(input)
>>> print(output)
Process with rank 0: [[0. 1.],
                      [2. 3.]]
Process with rank 1: [[4. 5.],
                      [6. 7.]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - CollectiveScatter

18.7.9 mindspore.ops.NeighborExchangeV2

class mindspore.ops.**NeighborExchangeV2**(*send_rank_ids*, *send_lens*, *recv_rank_ids*, *recv_lens*, *data_format*, *group=GlobalComm.WORLD_COMM_GROUP*)

NeighborExchangeV2 is a collective communication operation.

NeighborExchangeV2 sends data from the local rank to ranks in the *send_rank_ids*, as while receive data from *recv_rank_ids*. Please refer to the tutorial examples below to learn about how the data is exchanged between neighborhood devices.

Note: This operator requires a full-mesh network topology, each device has the same vlan id, and the ip & mask are in the same subnet, please check the details .

Users need to ensure that the length of the received data *recv_lens* is consistent with that of the sent data *send_lens*.

Parameters

- **send_rank_ids** (*list(int)*) −Ranks which the data is sent to. 8 rank_ids represents 8 directions, if one direction is not send to , set it -1.
- **recv_rank_ids** (*list(int)*) −Ranks which the data is received from. 8 rank_ids represents 8 directions, if one direction is not recv from , set it -1.
- **send_lens** (*list(int)*) −Data lens which send to the send_rank_ids, 4 numbers represent the lens of [send_top, send_bottom, send_left, send_right].
- **recv_lens** (*list(int)*) −Data lens which received from recv_rank_ids, 4 numbers represent the lens of [recv_top, recv_bottom, recv_left, recv_right].
- **data_format** (*str*) −Data format, only support NCHW now.
- **group** (*str, optional*) −The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP` , which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Inputs:

- **input_x** (Tensor) - The Tensor before being exchanged. It has a shape of (N, C, H, W).

Outputs:

The Tensor after being exchanged. If input shape is (N, C, H, W), output shape is $(N, C, H + recv\_top + recv\_bottom, W + recv\_left + recv\_right)$.

Raises

- **TypeError** −If *group* is not a string or any one of *send_rank_ids*, *recv_rank_ids*, *send_lens*, *recv_lens* is not a list.
- **ValueError** −If *send_rank_ids* or *recv_rank_ids* has value less than -1 or has repeated values.
- **ValueError** −If *send_lens*, *recv_lens* has value less than 0.
- **ValueError** −If *data_format* is not "NCHW".

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import os
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> import numpy as np
>>>
>>> class Net0(nn.Cell):
...     def __init__(self):
...         super(Net0, self).__init__()
...         self.neighbor_exchangev2 = ops.NeighborExchangeV2(send_rank_ids=[-1, -1, -1, -1,␣
↪1, -1, -1, -1],
...                                                           send_lens=[0, 1, 0, 0],
...                                                           recv_rank_ids=[-1, -1, -1, -1,␣
↪1, -1, -1, -1],
...                                                           recv_lens=[0, 1, 0, 0], data_
↪format="NCHW")
...
...     def construct(self, x):
...         out = self.neighbor_exchangev2(x)
...         return out
>>> class Net1(nn.Cell):
...     def __init__(self):
...         super(Net1, self).__init__()
...         self.neighbor_exchangev2 = ops.NeighborExchangeV2(send_rank_ids=[0, -1, -1, -1, -
↪1, -1, -1, -1],
...                                                           send_lens=[1, 0, 0, 0],
...                                                           recv_rank_ids=[0, -1, -1, -1, -
↪1, -1, -1, -1],
...                                                           recv_lens=[1, 0, 0, 0], data_
↪format="NCHW")
...
...     def construct(self, x):
...         out = self.neighbor_exchangev2(x)
...         return out
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
>>> rank_id = int(os.getenv("RANK_ID"))
>>> if (rank_id % 2 == 0):
...     input_x = ms.Tensor(np.ones([1, 1, 2, 2]), dtype = ms.float32)
...     net = Net0()
...     output = net(input_x)
...     print(output)
... else:
...     input_x = ms.Tensor(np.ones([1, 1, 2, 2]) * 2, dtype = ms.float32)
...     net = Net1()
```

(continues on next page)

(continued from previous page)

```
...     output = net(input_x)
...     print(output)
rank 0:
[[[[1. 1.], [1. 1.], [2. 2.]]]]
rank 1:
[[[[1. 1.], [2. 2.], [2. 2.]]]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - NeighborExchangeV2

18.7.10 mindspore.ops.Receive

class mindspore.ops.**Receive**(*sr_tag*, *src_rank*, *shape*, *dtype*, *group=GlobalComm.WORLD_COMM_GROUP*, *group_back=GlobalComm.WORLD_COMM_GROUP*)

Receive tensors from src_rank.

Note: Send and Receive must be used in combination and have same *sr_tag*.

Parameters

- **sr_tag** (*int*) – A required integer identifying the send/recv message tag. This operator will receive the tensor sent by the Send operator with the same *sr_tag* tag.
- **src_rank** (*int*) – A required integer identifying the source rank.
- **shape** (*list[int]*) – A required list identifying the shape of the tensor to be received.
- **dtype** (*Type*) – A required Type identifying the type of the tensor to be received. The supported types: int8/int16/int32/float16/float32.
- **group** (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.
- **group_back** (*str, optional*) – The communication group for backpropagation. Default: `GlobalComm.WORLD_COMM_GROUP`.

Outputs:

Tensor, output has the same shape as the Tensor sent by *Send* operation.

Raises

- **TypeError** – If *src_rank* is not an int or *group* is not a str.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.
- **ValueError** – If the local rank id of the calling process in the group is larger than the group's rank size.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import os
>>> import numpy as np
>>> import mindspore.ops as ops
>>> import mindspore.nn as nn
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE, jit_level="O2")
>>> init()
>>>
>>> class SendNet(nn.Cell):
...     def __init__(self):
...         super(SendNet, self).__init__()
...         self.depend = ops.Depend()
...         self.send = ops.Send(sr_tag=0, dest_rank=1, group="hccl_world_group")
...
...     def construct(self, x):
...         out = self.depend(x, self.send(x))
...         return out
>>>
>>> class ReceiveNet(nn.Cell):
...     def __init__(self):
...         super(ReceiveNet, self).__init__()
...         self.recv = ops.Receive(sr_tag=0, src_rank=0, shape=[2, 8], dtype=ms.float32,
...                                 group="hccl_world_group")
...
...     def construct(self):
...         out = self.recv()
...         return out
>>>
>>> if __name__ == "__main__":
...     rank_id = os.environ["RANK_ID"]
...     rank_size = os.environ["RANK_SIZE"]
...     if rank_id == "0":
...         input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...         send_net = SendNet()
...         output = send_net(input_)
...     else:
...         recv_net = ReceiveNet()
...         output = recv_net()
...         print(output.asnumpy())
```

Tutorial Examples:

- Distributed Set Communication Primitives - Receive

18.7.11 mindspore.ops.ReduceOp

class mindspore.ops.**ReduceOp**

Operation options for reducing tensors. This is an enumerated type, not an operator.

The main calling methods are as follows:

- SUM: ReduceOp.SUM.
- MAX: ReduceOp.MAX.
- MIN: ReduceOp.MIN.
- PROD: ReduceOp.PROD.

There are four kinds of operation options, "SUM", "MAX", "MIN", and "PROD".

- SUM: Take the sum.
- MAX: Take the maximum.
- MIN: Take the minimum.
- PROD: Take the product.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import numpy as np
>>> import mindspore
>>> from mindspore.communication import init
>>> from mindspore import Tensor, ops, nn
>>> from mindspore.ops import ReduceOp
>>>
>>> init()
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.allreduce_sum = ops.AllReduce(ReduceOp.SUM)
...
...     def construct(self, x):
...         return self.allreduce_sum(x)
...
>>> input_ = Tensor(np.ones([2, 8]).astype(np.float32))
>>> net = Net()
>>> output = net(input_)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

18.7.12 mindspore.ops.ReduceScatter

class mindspore.ops.**ReduceScatter**(*op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*)

Reduces and scatters tensors from the specified communication group and returns the tensor which is reduced and scattered.

Note: The tensors must have the same shape and format in all processes of the collection.

Parameters

- **op** (*str, optional*) – Specifies an operation used for element-wise reductions, like SUM and MAX. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.

Inputs:

- **input_x** (Tensor) - Input Tensor, suppose it has a shape $(N, *)$, where * means any number of additional dimensions. N must be divisible by rank_size. rank_size refers to the number of cards in the communication group.

Outputs:

Tensor, it has the same dtype as *input_x* with a shape of $(N/rank\_size, *)$.

Raises

- **TypeError** – If any of operation and group is not a string.
- **ValueError** – If the first dimension of the input cannot be divided by the rank_size.
- **RuntimeError** – If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.communication import init
>>> from mindspore.ops import ReduceOp
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> import numpy as np
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> init()
```

(continues on next page)

(continued from previous page)

```
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.reducescatter = ops.ReduceScatter(ReduceOp.SUM)
...
...     def construct(self, x):
...         return self.reducescatter(x)
...
>>> input_ = Tensor(np.ones([8, 8]).astype(np.float32))
>>> net = Net()
>>> output = net(input_)
>>> print(output)
[[2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2.]]
```

Tutorial Examples:

- Distributed Set Communication Primitives - ReduceScatter

18.7.13 mindspore.ops.Reduce

class mindspore.ops.**Reduce**(*dest_rank*, *op=ReduceOp.SUM*, *group=GlobalComm.WORLD_COMM_GROUP*)

Reduces tensors across the processes in the specified communication group, sends the result to the target dest_rank(local rank), and returns the tensor which is sent to the target process.

Note: Only process with destination rank receives the reduced output. Support PyNative mode and Graph mode, but Graph mode only supports scenes with a graph compilation level of O0. Other processes only get a tensor with shape [1], which has no mathematical meaning.

Parameters

- **dest_rank** (*int*) –The target process(local rank) in the specific group that receives the reduced output.
- **op** (*str, optional*) –Specifies an operation used for element-wise reductions, like sum, prod, max, and min. On the CPU, only 'sum' is supported. Default: `ReduceOp.SUM`.
- **group** (*str, optional*) –The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`, which means `"hccl_world_group"` in Ascend, and `"nccl_world_group"` in GPU.

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Outputs:

Tensor. Return the tensor in the specific rank of the process after reduction. The shape of tensor is $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** –If the type of the first input parameter is not Tensor, or any of *op* and *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.

Supported Platforms:

`Ascend`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 4 devices.

```
>>> from mindspore import ops
>>> import mindspore.nn as nn
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>> import numpy as np
>>> # Launch 4 processes.
>>> init()
>>> class ReduceNet(nn.Cell):
...     def __init__(self):
...         super(ReduceNet, self).__init__()
...         self.reduce = ops.Reduce(dest_rank=1)
...
...     def construct(self, x):
...         out = self.reduce(x)
...         return out
>>> input = Tensor(np.ones([2, 8]).astype(np.float32))
>>> net = ReduceNet()
>>> output = net(input)
>>> print(output)
Process with rank 1: [[4. 4. 4. 4. 4. 4. 4. 4.]
                      [4. 4. 4. 4. 4. 4. 4. 4.]],
Other proesses: [0.].
```

18.7.14 mindspore.ops.Send

class mindspore.ops.**Send**(*sr_tag*, *dest_rank*, *group=GlobalComm.WORLD_COMM_GROUP*, *group_back=GlobalComm.WORLD_COMM_GROUP*)

Send tensors to the specified dest_rank.

Note: Send and Receive must be used in combination and have same *sr_tag*.

Parameters

- **sr_tag** (*int*) – The tag to identify the send/recv message. The message sent by this operator will be received by the Receive op with the same "sr_tag".
- **dest_rank** (*int*) – A required integer identifying the destination rank.
- **group** (*str*, *optional*) – The communication group to work on. Default: `GlobalComm.WORLD_COMM_GROUP`.

- **group_back** (*str, optional*) –The communication group for backpropagation. Default: `GlobalComm.WORLD_COMM_GROUP`.

Inputs:

- **input_x** (Tensor) - The shape of tensor is $(x_1, x_2, ..., x_R)$.

Raises

- **TypeError** –If *group* is not a str.
- **RuntimeError** –If device target is invalid, or backend is invalid, or distributed initialization fails.
- **ValueError** –If the local rank id of the calling process in the group is larger than the group's rank size.

Supported Platforms:

`Ascend` `GPU`

Examples

Note: Before running the following examples, you need to configure the communication environment variables.

For Ascend/GPU/CPU devices, it is recommended to use the msrun startup method without any third-party or configuration file dependencies. Please see the msrun start up for more details.

This example should be run with 2 devices.

```
>>> import os
>>> import numpy as np
>>> import mindspore.ops as ops
>>> import mindspore.nn as nn
>>> import mindspore as ms
>>> from mindspore.communication import init
>>> from mindspore import Tensor
>>>
>>> ms.set_context(mode=ms.GRAPH_MODE, jit_level="O2")
>>> init()
>>>
>>> class SendNet(nn.Cell):
...     def __init__(self):
...         super(SendNet, self).__init__()
...         self.depend = ops.Depend()
...         self.send = ops.Send(sr_tag=0, dest_rank=1, group="hccl_world_group")
...
...     def construct(self, x):
...         out = self.depend(x, self.send(x))
...         return out
>>>
>>> class ReceiveNet(nn.Cell):
...     def __init__(self):
...         super(ReceiveNet, self).__init__()
...         self.recv = ops.Receive(sr_tag=0, src_rank=0, shape=[2, 8], dtype=ms.float32,
...                                 group="hccl_world_group")
...
...     def construct(self):
```

(continues on next page)

(continued from previous page)

```
...         out = self.recv()
...         return out
>>>
>>> if __name__ == "__main__":
...     rank_id = os.environ["RANK_ID"]
...     rank_size = os.environ["RANK_SIZE"]
...     if rank_id == "0":
...         input_ = Tensor(np.ones([2, 8]).astype(np.float32))
...         send_net = SendNet()
...         output = send_net(input_)
...     else:
...         recv_net = ReceiveNet()
...         output = recv_net()
...         print(output.asnumpy())
```

Tutorial Examples:

- Distributed Set Communication Primitives - Send

18.8 Debugging Operator

API Name	Description	Supported Platforms
mindspore.ops.HistogramSummary	This operator will calculate the histogram of a tensor and put it to a summary file with protocol buffer format.	Ascend GPU CPU
mindspore.ops.ImageSummary	This operator will put an image tensor to a summary file with protocol buffer format.	Ascend GPU CPU
mindspore.ops.ScalarSummary	This operator will put a scalar to a summary file with protocol buffer format.	Ascend GPU CPU
mindspore.ops.TensorSummary	This operator will put a tensor to a summary file with protocol buffer format.	Ascend GPU CPU
mindspore.ops.TensorDump	Save the Tensor as an npy file in numpy format.	Ascend
mindspore.ops.Print	Print the inputs to stdout.	Ascend GPU CPU
mindspore.ops.NPUAllocFloatStatus	Allocates a flag to store the overflow status.	Ascend
mindspore.ops.NPUClearFloatStatus	Clears the flag which stores the overflow status.	Ascend
mindspore.ops.NPUGetFloatStatus	*mindspore.ops.NPUGetFloatStatus* updates the flag which is the output tensor of *mindspore.ops.NPUAllocFloatStatus* with the latest overflow status.	Ascend

18.8.1 mindspore.ops.HistogramSummary

class mindspore.ops.**HistogramSummary**

This operator will calculate the histogram of a tensor and put it to a summary file with protocol buffer format. It must be used with SummaryRecord or SummaryCollector, which specify the directory of the summary file. In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when calling this operator intensively.

Inputs:

- **name** (str) - The name of the input variable.

- **value** (Tensor) - The value of tensor, and the rank of tensor must be greater than 0.

Raises

- **TypeError** – If *name* is not a str.
- **TypeError** – If *value* is not a Tensor.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import Tensor, set_context
>>>
>>>
>>> class SummaryDemo(nn.Cell):
...     def __init__(self,):
...         super(SummaryDemo, self).__init__()
...         self.summary = ops.HistogramSummary()
...         self.add = ops.Add()
...
...     def construct(self, x, y):
...         x = self.add(x, y)
...         name = "x"
...         self.summary(name, x)
...         return x
>>> set_context(mode=mindspore.GRAPH_MODE)
>>> summary = SummaryDemo()(Tensor([1, 2]), Tensor([3, 4]))
>>> print(summary)
[4 6]
```

18.8.2 mindspore.ops.ImageSummary

class mindspore.ops.**ImageSummary**

This operator will put an image tensor to a summary file with protocol buffer format. It must be used with SummaryRecord or SummaryCollector, which specify the directory of the summary file. In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve execution failure when calling this operator intensively.

Inputs:

- **name** (str) - The name of the input variable, it must not be an empty string.
- **value** (Tensor) - The value of image, the rank of tensor must be 4.

Raises

- **TypeError** – If *name* is not a str.
- **TypeError** – If *value* is not a Tensor.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>>
>>>
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.summary = ops.ImageSummary()
...
...     def construct(self, x):
...         name = "image"
...         self.summary(name, x)
...         return x
...
```

18.8.3 mindspore.ops.ScalarSummary

class mindspore.ops.**ScalarSummary**

This operator will put a scalar to a summary file with protocol buffer format. It must be used with *mindspore.SummaryRecord* or *mindspore.SummaryCollector*, which specify the directory of the summary file. In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when calling this operator intensively.

Inputs:

- **name** (str) - The name of the input variable, it must not be an empty string.
- **value** (Tensor) - The value of scalar, and the dim of *value* must be 0 or 1.

Raises

- **TypeError** – If *name* is not a str.
- **TypeError** – If *value* is not a Tensor.
- **ValueError** – If dim of *value* is greater than 1.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import Tensor, set_context
>>>
>>>
>>> class SummaryDemo(nn.Cell):
...     def __init__(self,):
```

(continues on next page)

(continued from previous page)

```
...         super(SummaryDemo, self).__init__()
...         self.summary = ops.ScalarSummary()
...         self.add = ops.Add()
...
...     def construct(self, x, y):
...         name = "x"
...         self.summary(name, x)
...         x = self.add(x, y)
...         return x
>>> set_context(mode=mindspore.GRAPH_MODE)
>>> summary = SummaryDemo()(Tensor(3), Tensor(4))
>>> print(summary)
7
```

18.8.4 mindspore.ops.TensorSummary

class mindspore.ops.**TensorSummary**

This operator will put a tensor to a summary file with protocol buffer format. It must be used with SummaryRecord or SummaryCollector, which specify the directory of the summary file. In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when calling this operator intensively.

Inputs:

- **name** (str) - The name of the input variable.
- **value** (Tensor) - The value of tensor, and the rank of tensor must be greater than 0.

Raises

- **TypeError** – If *name* is not a str.
- **TypeError** – If *value* is not a Tensor.
- **ValueError** – If rank of *value* is 0.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import Tensor, set_context
>>>
>>>
>>> class SummaryDemo(nn.Cell):
...     def __init__(self,):
...         super(SummaryDemo, self).__init__()
...         self.summary = ops.TensorSummary()
...         self.add = ops.Add()
...
...     def construct(self, x, y):
```

(continues on next page)

(continued from previous page)

```
...         x = self.add(x, y)
...         name = "x"
...         self.summary(name, x)
...         return x
>>> set_context(mode=mindspore.GRAPH_MODE)
>>> summary = SummaryDemo()(Tensor([[1]]), Tensor([[2]]))
>>> print(summary)
[[3]]
```

18.8.5 mindspore.ops.TensorDump

class mindspore.ops.**TensorDump**(*input_output='out'*)

Save the Tensor as an npy file in numpy format.

> **Warning:** The parameter input_output will no longer support the value 'all'.

Note: In Ascend platform with graph mode, the environment variables *MS_DUMP_SLICE_SIZE* and *MS_DUMP_WAIT_TIME* can be set to solve operator execution failure when outputting big tensor or outputting tensor intensively.

Parameters

input_output (*str, optional*) – Used to control Tensordump behavior. Available value is one of ['in', 'out']. Default value is `out`.

In case of OpA –> RedistributionOps –> OpB, The dump data of OpA's output is not equal to OpB's input (Due to the redistribution operators). So the parameter input_output is to handle this situation.

Assuming OpA's output is used as both Tensordump's input parameter and OpB's input parameter. Different requirements of saving dump data can be achieved by configuring parameter input_output:

- If the input_output is 'out', the dump data contains only OpA's output slice.
- If the input_output is 'in', the dump data contains only OpB's input slice.

For input_output is 'in', the input slice npy file format is: fileName_dumpMode_dtype_id.npy.

For input_output is 'out', the output slice npy file format is: fileName_dtype_id.npy.

- fileName: Value of the parameter file (if parameter file_name is a user-specified path, the value of fileName is the last level of the path).
- dumpMode: Value of the parameter input_output.
- dtype: The original data type. Data of type bfloat16 stored in the .npy file will be converted to float32.
- id: An auto increment ID.

Inputs:

- **file** (str) - The path of the file to be saved.
- **input_x** (Tensor) - Input Tensor of any dimension.

Raises

- **TypeError** – If *file* is not a str.
- **TypeError** – If *input_x* is not a Tensor.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> import time
>>> from mindspore import nn, Tensor, ops
>>> ms.set_context(mode=ms.GRAPH_MODE)
>>> ms.set_device(device_target="Ascend")
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.dump = ops.TensorDump()
...
...     def construct(self, x):
...         x += 1.
...         self.dump('add', x)
...         x /= 2.
...         self.dump('div', x)
...         x *= 5.
...         self.dump('mul', x)
...         return x
...
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]).astype(np.float32)
>>> input_x = Tensor(x)
>>> net = Net()
>>> out = net(input_x)
>>> time.sleep(0.5)
>>> add = np.load('add_float32_0.npy')
>>> print(add)
[[2. 3. 4. 5.]
 [6. 7. 8. 9.]]
```

18.8.6 mindspore.ops.Print

class mindspore.ops.**Print**

Print the inputs to stdout.

Refer to *mindspore.ops.print_()* for more detail.

Inputs:

- **input_x** (Union[Tensor, bool, int, float, str]) - The graph node to attach to. Supports multiple inputs which are separated by ','.

Outputs:

Tensor, has the same data type and shape as original *input_x*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, nn, ops
>>> class PrintDemo(nn.Cell):
...     def __init__(self):
...         super(PrintDemo, self).__init__()
...         self.print = ops.Print()
...
...     def construct(self, x, y):
...         self.print('Print Tensor x and Tensor y:', x, y)
...         return x
...
>>> x = Tensor(np.ones([2, 1]).astype(np.int32))
>>> y = Tensor(np.ones([2, 2]).astype(np.int32))
>>> net = PrintDemo()
>>> result = net(x, y)
Print Tensor x and Tensor y:
Tensor(shape=[2, 1], dtype=Int32, value=
[[1],
 [1]])
Tensor(shape=[2, 2], dtype=Int32, value=
[[1, 1],
 [1, 1]])
```

18.8.7 mindspore.ops.NPUAllocFloatStatus

class mindspore.ops.**NPUAllocFloatStatus**

Allocates a flag to store the overflow status.

The flag is a tensor whose shape is $(8,)$ and data type is *mindspore.dtype.float32*.

Note: Please refer to the Examples of *mindspore.ops.NPUGetFloatStatus*.

Outputs:

Tensor, has the shape of $(8,)$.

Supported Platforms:

Ascend

Examples

```
>>> from mindspore import ops
>>> alloc_status = ops.NPUAllocFloatStatus()
>>> output = alloc_status()
>>> print(output)
[0. 0. 0. 0. 0. 0. 0. 0.]
```

18.8.8 mindspore.ops.NPUClearFloatStatus

class mindspore.ops.**NPUClearFloatStatus**

Clears the flag which stores the overflow status.

Note: The flag is in the register on the *Ascend* device. It will be reset and can not be reused again after the *NPUClearFloatStatus* is called. In addition, there are strict sequencing requirements for use, i.e., before using the NPUGetFloatStatus operator, need to ensure that the NPUClearFlotStatus and your compute has been executed. We use *mindspore.ops.Depend* on ensure the execution order.

Please refer to the Examples of *mindspore.ops.NPUGetFloatStatus*.

Inputs:

- **x** (Tensor) - The output tensor of *NPUAllocFloatStatus*. The data type must be float16 or float32.

Outputs:

Tensor, has the same shape as *x*. All the elements in the tensor will be zero.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.alloc_status = ops.NPUAllocFloatStatus()
...         self.get_status = ops.NPUGetFloatStatus()
...         self.clear_status = ops.NPUClearFloatStatus()
...         self.sub = ops.Sub()
...         self.neg = ops.Neg()
...
...     def construct(self, x):
...         init = self.alloc_status()
...         clear_status = self.clear_status(init)
...         x = ops.depend(x, clear_status)
...         res = self.sub(x, self.neg(x))
...         init = ops.depend(init, res)
```

(continues on next page)

(continued from previous page)

```
...         get_status = self.get_status(init)
...         res = ops.depend(res, get_status)
...         return res
>>>
>>> value = 5
>>> data = np.full((2, 3), value, dtype=np.float16)
>>> x = Tensor(data, dtype=mstype.float16)
>>> net = Net()
>>> res = net(x)
>>> print(res)
[[10. 10. 10.]
 [10. 10. 10.]]
```

18.8.9 mindspore.ops.NPUGetFloatStatus

class mindspore.ops.**NPUGetFloatStatus**

mindspore.ops.NPUGetFloatStatus updates the flag which is the output tensor of *mindspore.ops.NPUAllocFloatStatus* with the latest overflow status.

Note: The flag is a tensor whose shape is $(8,)$ and data type is *mindspore.dtype.float32*. If the sum of the flag equals to 0, there is no overflow happened. If the sum of the flag is bigger than 0, there is overflow happened. In addition, there are strict sequencing requirements for use, i.e., before using the NPUGetFloatStatus operator, need to ensure that the NPUClearFlotStatus and your compute has been executed. We use *mindspore.ops.Depend* to ensure the correct execution order.

Inputs:

- **x** (Tensor) - The output tensor of *NPUAllocFloatStatus*. The data type must be float16 or float32. $(N, *)$ where $*$ means, any number of additional dimensions, its rank should be less than 8.

Outputs:

Tensor, has the same shape as *x*.

Raises

- **TypeError** – If *x* is not a Tensor.
- **TypeError** – If dtype of *x* is neither float16 nor float32.

Supported Platforms:

`Ascend`

Examples

```
>>> import numpy as np
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def __init__(self):
```

(continues on next page)

(continued from previous page)

```
...         super().__init__()
...         self.alloc_status = ops.NPUAllocFloatStatus()
...         self.get_status = ops.NPUGetFloatStatus()
...         self.clear_status = ops.NPUClearFloatStatus()
...         self.sub = ops.Sub()
...         self.neg = ops.Neg()
...
...     def construct(self, x):
...         init = self.alloc_status()
...         clear_status = self.clear_status(init)
...         x = ops.depend(x, clear_status)
...         res = self.sub(x, self.neg(x))
...         init = ops.depend(init, res)
...         get_status = self.get_status(init)
...         res = ops.depend(res, get_status)
...         return res
>>>
>>> value = 5
>>> data = np.full((2, 3), value, dtype=np.float16)
>>> x = Tensor(data, dtype=mstype.float16)
>>> net = Net()
>>> res = net(x)
>>> print(res)
[[10. 10. 10.]
 [10. 10. 10.]]
```

18.9 Sparse Operator

API Name	Description	Supported Platforms
mindspore.ops.SparseTensorDenseMatmul	Multiplies sparse matrix *A* by dense matrix *B*.	GPU CPU
mindspore.ops.SparseToDense	Converts a sparse representation into a dense tensor.	CPU

18.9.1 mindspore.ops.SparseTensorDenseMatmul

class mindspore.ops.**SparseTensorDenseMatmul**(*adjoint_st=False*, *adjoint_dt=False*)

Multiplies sparse matrix *A* by dense matrix *B*. The rank of sparse matrix and dense matrix must be equal to *2*.

Parameters

- **adjoint_st** (*bool*) –If `True` , sparse tensor is transposed before multiplication. Default: `False` .
- **adjoint_dt** (*bool*) –If `True` , dense tensor is transposed before multiplication. Default: `False` .

Inputs:

- **indices** (Tensor) - A 2-D Tensor, represents the position of the element in the sparse tensor. Support int32, int64, each element value should be a non-negative int number. The shape is $(n, 2)$.
- **values** (Tensor) - A 1-D Tensor, represents the value corresponding to the position in the *indices*. Support float16, float32, float64, int32, int64, complex64, complex128. The shape should be $(n,)$.

- **sparse_shape** (tuple(int) or (Tensor)) - A positive int tuple or tensor which specifies the shape of sparse tensor, and only constant value is allowed when sparse_shape is a tensor, should have 2 elements, represent sparse tensor shape is (N, C).
- **dense** (Tensor) - A 2-D Tensor, the dtype is same as *values*. If *adjoint_st* is False and *adjoint_dt* is False, the shape must be (C, M). If *adjoint_st* is False and *adjoint_dt* is True, the shape must be (M, C). If *adjoint_st* is True and *adjoint_dt* is False, the shape must be (N, M). If *adjoint_st* is True and *adjoint_dt* is True, the shape must be (M, N).

Outputs:

Tensor, the dtype is the same as *values*. If *adjoint_st* is False, the shape is (N, M). If *adjoint_st* is True, the shape is (C, M).

Raises

- **TypeError** –If the type of *adjoint_st* or *adjoint_dt* is not bool, or the dtype of *indices*, dtype of *values* and dtype of *dense* don't meet the parameter description.
- **ValueError** –If the shapes of *sparse_shape*, *indices*, *values*, and *dense* don't meet the constraints in the parameter description.

Supported Platforms:

GPU CPU

Examples

```
>>> import mindspore
>>> from mindspore import Tensor
>>> from mindspore.ops import operations as ops
>>> from mindspore import dtype as mstype
>>> indices = Tensor([[0, 1], [1, 2]], dtype=mindspore.int32)
>>> values = Tensor([1, 2], dtype=mindspore.float32)
>>> sparse_shape = (3, 4)
>>> dense = Tensor([[1, 1], [2, 2], [3, 3], [4, 4]], dtype=mindspore.float32)
>>> sparse_dense_matmul = ops.SparseTensorDenseMatmul()
>>> out = sparse_dense_matmul(indices, values, sparse_shape, dense)
>>> print(out)
[[2. 2.]
 [6. 6.]
 [0. 0.]]
```

18.9.2 mindspore.ops.SparseToDense

class mindspore.ops.**SparseToDense**

Converts a sparse representation into a dense tensor.

Inputs:

- **indices** (Tensor) - A 2-D Tensor, represents the position of the element in the sparse tensor. Support int32, int64, each element value should be a non-negative int number. The shape is $(n, 2)$.
- **values** (Tensor) - A 1-D Tensor, represents the value corresponding to the position in the *indices*. The shape should be $(n,)$.
- **sparse_shape** (tuple(int)) - A positive int tuple which specifies the shape of sparse tensor, should have 2 elements, represent sparse tensor shape is (N, C).

Outputs:

Tensor, converted from sparse tensor. The dtype is same as *values*, and the shape is *sparse_shape*.

Raises

- **TypeError** – If the dtype of *indices* is neither int32 nor int64.
- **ValueError** – If *sparse_shape*, shape of *indices* and shape of *values* don't meet the parameter description.

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> indices = Tensor([[0, 1], [1, 2]])
>>> values = Tensor([1, 2], dtype=mindspore.float32)
>>> sparse_shape = (3, 4)
>>> sparse_to_dense = ops.SparseToDense()
>>> out = sparse_to_dense(indices, values, sparse_shape)
>>> print(out)
[[0. 1. 0. 0.]
 [0. 0. 2. 0.]
 [0. 0. 0. 0.]]
```

18.10 Frame Operators

API Name	Description	Supported Platforms
mindspore.ops.Depend	Depend is used for processing dependency operations.	Ascend GPU CPU
mindspore.ops.ForiLoop	Performs a loop operation within the specified range.	Ascend GPU CPU
mindspore.ops.GradOperation	A higher-order function which is used to generate the gradient function for the input function.	Ascend GPU CPU
mindspore.ops.HookBackward	This operation is used as a tag to hook gradient in intermediate variables.	Ascend GPU CPU
mindspore.ops.HyperMap	HyperMap will apply the set operation to input sequences.	Ascend GPU CPU
mindspore.ops.InsertGradientOf	Attaches callback to the graph node that will be invoked on the node's gradient.	Ascend GPU CPU
mindspore.ops.Morph	The *Morph* Primitive is used to encapsulate a user-defined function *fn*, allowing it to be used as a custom Primitive.	
mindspore.ops.Map	Map will apply the set operation on input sequences.	Ascend GPU CPU
mindspore.ops.MultitypeFuncGraph	MultitypeFuncGraph is a class used to generate overloaded functions, considering different types as inputs.	Ascend GPU CPU
mindspore.ops.Partial	Makes a partial function instance.	Ascend GPU CPU

continues on next page

Table 24 – continued from previous page

mindspore.ops.Scan	Scan a function over an array while the processing of the current element depends on the execution result of the previous element.	Ascend GPU CPU
mindspore.ops.WhileLoop	Provide a useful op for reducing compilation times of while loop.	Ascend GPU CPU

18.10.1 mindspore.ops.Depend

class mindspore.ops.**Depend**

Depend is used for processing dependency operations.

In most scenarios, if operators have IO side effects or memory side effects, they will be executed according to the user's semantics. In some scenarios, if the two operators A and B have no order dependency, and A must be executed before B, we recommend using Depend to specify their execution order. The usage method is as follows:

```
a = A(x)                    --->        a = A(x)
b = B(y)                    --->        y = Depend(y, a)
                            --->        b = B(y)
```

Inputs:

- **value** (Tensor) - the real value to return for depend operator.
- **expr** (Expression) - the expression to execute with no outputs.

Outputs:

Tensor, the value passed by last operator.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.softmax = ops.Softmax()
...         self.depend = ops.Depend()
...
...     def construct(self, x, y):
...         mul = x * y
...         y = self.depend(y, mul)
...         ret = self.softmax(y)
...         return ret
...
>>> x = Tensor(np.ones([4, 5]), dtype=mindspore.float32)
>>> y = Tensor(np.ones([4, 5]), dtype=mindspore.float32)
>>> net = Net()
```

(continues on next page)

(continued from previous page)

```
>>> output = net(x, y)
>>> print(output)
[[0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]
 [0.2 0.2 0.2 0.2 0.2]]
```

18.10.2 mindspore.ops.ForiLoop

class mindspore.ops.**ForiLoop**

Performs a loop operation within the specified range. The execution logic of the ForiLoop operator can be roughly represented by the following code:

```
def ForiLoop(lower, upper, loop_func, init_val):
    for i in range(lower, upper):
        init_val = loop_func(i, init_val)
    return init_val
```

The current ForiLoop operator has the following syntactic limitations:

- Using a side-effect function as *loop_func* is currently not support, such as operations that modify parameters, global variables, etc.
- The return value of *loop_func* being of a different type or shape from the *init_val* is currently not support.
- Negative numbers or custom increments is currently not support.

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **lower** (Union[int, Tensor]) - The start index of loop.
- **upper** (Union[int, Tensor]) - The end index of loop.
- **loop_func** (Function) - The loop function, takes two arguments.
- **init_val** (Union[Tensor, number, str, bool, list, tuple, dict]) - The init value.
- **unroll** (bool, optional) - The flag for whether unroll in compile process, only valid when the number of loop iterations is determined. Default: `True` .

Outputs:

Union[Tensor, number, str, bool, list, tuple, dict], the final result of the loop, has same type and shape with input *init_val* .

Raises

- **TypeError** – If *lower* is not an int or a Tensor.
- **TypeError** – If *upper* is not an int or a Tensor.
- **TypeError** – If *loop_func* is not a function.
- **ValueError** – If *loop_func* cannot take index and *init_val* as arguments or if the type of output it produces is different from the type or shape of *init_val* .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> def cumsum(index, res):
...     return index + res
...
>>> result_init = 0
>>> fori_loop = ops.ForiLoop()
>>> result = fori_loop(0, 4, cumsum, result_init)
>>> print(result)
6
```

18.10.3 mindspore.ops.GradOperation

class mindspore.ops.**GradOperation**(*get_all=False*, *get_by_list=False*, *sens_param=False*)

A higher-order function which is used to generate the gradient function for the input function.

The gradient function generated by *GradOperation* higher-order function can be customized by construction arguments.

For example, given an input function *net = Net()* that takes *x* and *y* as inputs, and has a parameter *z*, see *Net* in Examples.

- Used to get the derivative of the input:
  1. Returns gradients with respect to the first input (see *GradNetWrtX* in Examples).
     1) Construct a *GradOperation* higher-order function with default arguments: *grad_op = GradOperation()*.
     2) Call it with input function as argument to get the gradient function: *gradient_function = grad_op(net)*.
     3) Call the gradient function with input function's inputs to get the gradients with respect to the first input: *grad_op(net)(x, y)*.
  2. Returns gradients with respect to all inputs (see *GradNetWrtXY* in Examples).
     1) Construct a *GradOperation* higher-order function with *get_all=True* which indicates getting gradients with respect to all inputs, they are *x* and *y* in example function *Net()*: *grad_op = GradOperation(get_all=True)*.
     2) Call it with input function as argument to get the gradient function: *gradient_function = grad_op(net)*.
     3) Call the gradient function with input function's inputs to get the gradients with respect to all inputs: *gradient_function(x, y)*.
- Used to get the derivative of the parameters:

  Returns gradients with respect to given parameters (see *GradNetWithWrtParams* in Examples).
  1. Construct a *GradOperation* higher-order function with *get_by_list=True*: *grad_op = GradOperation(get_by_list=True)*.
  2. Construct a *ParameterTuple* that will be passed to the input function when constructing *GradOperation* higher-order function, it will be used as a parameter filter that determine which gradient to return: *params = ParameterTuple(net.trainable_params())*.
  3. Call it with input function and *params* as arguments to get the gradient function: *gradient_function = grad_op(net, params)*.
  4. Call the gradient function with input function's inputs to get the gradients with respect to given parameters: *gradient_function(x, y)*.

- Used to get the derivative of the inputs and parameters at the same time: Returns gradients with respect to all inputs and given parameters in the format of ((dx, dy), (dz)) (see *GradNetWrtInputsAndParams* in Examples).
    1. Construct a *GradOperation* higher-order function with *get_all=True* and *get_by_list=True*: *grad_op = GradOperation(get_all=True, get_by_list=True)*.
    2. Construct a *ParameterTuple* that will be passed along input function when constructing *GradOperation* higher-order function: *params = ParameterTuple(net.trainable_params())*.
    3. Call it with input function and *params* as arguments to get the gradient function: *gradient_function = grad_op(net, params)*.
    4. Call the gradient function with input function's inputs to get the gradients with respect to all inputs and given parameters: *gradient_function(x, y)*.
- We can configure the sensitivity(gradient with respect to output) by setting *sens_param* as True and passing an extra sensitivity input to the gradient function, the sensitivity input should has the same shape and type with input function's output(see *GradNetWrtXYWithSensParam* in Examples).
    1. Construct a *GradOperation* higher-order function with *get_all=True* and *sens_param=True*: *grad_op = GradOperation(get_all=True, sens_param=True)*.
    2. Define *grad_wrt_output* as *sens_param* which works as the gradient with respect to output: *grad_wrt_output = Tensor(np.ones([2, 2]).astype(np.float32))*.
    3. Call it with input function as argument to get the gradient function: *gradient_function = grad_op(net)*.
    4. Call the gradient function with input function's inputs and *sens_param* to get the gradients with respect to all inputs: *gradient_function(x, y, grad_wrt_output)*.

Note: For above gradient functions, the returned gradient result may vary for grad result element number:

- Return a single value if only one result.
- Return a tuple for multiple results.
- Return an empty tuple for no result.

Parameters

- **get_all** (*bool*) – If `True` , get all the gradients with respect to inputs. Default: `False` .
- **get_by_list** (*bool*) – If `True` , get all the gradients with respect to Parameter free variables. If get_all and get_by_list are both `False` , get the gradient with respect to first input. If get_all and get_by_list are both `True` , get the gradients with respect to inputs and Parameter free variables at the same time in the form of ("gradients with respect to inputs", "gradients with respect to parameter free variables"). Default: `False` .
- **sens_param** (*bool*) – Whether to append sensitivity (gradient with respect to output) as input. If sens_param is `False` , a 'ones_like(outputs)' sensitivity will be attached automatically. Default: `False` . If the sensor_param is `True` , a sensitivity (gradient with respect to output) needs to be transferred through the positional parameter or key-value pair parameter. If the value is transferred through the key-value pair parameter, the key must be sens.

Returns

The higher-order function which takes a function as argument and returns gradient function for it.

Raises

TypeError – If *get_all*, *get_by_list* or *sens_param* is not a bool.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import numpy as np
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops, nn, Parameter
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.matmul = ops.MatMul()
...         self.z = Parameter(Tensor(np.array([1.0], np.float32)), name='z')
...     def construct(self, x, y):
...         x = x * self.z
...         out = self.matmul(x, y)
...         return out
...
>>> class GradNetWrtX(nn.Cell):
...     def __init__(self, net):
...         super(GradNetWrtX, self).__init__()
...         self.net = net
...         self.grad_op = ops.GradOperation()
...     def construct(self, x, y):
...         gradient_function = self.grad_op(self.net)
...         return gradient_function(x, y)
...
>>> x = Tensor([[0.5, 0.6, 0.4], [1.2, 1.3, 1.1]], dtype=mstype.float32)
>>> y = Tensor([[0.01, 0.3, 1.1], [0.1, 0.2, 1.3], [2.1, 1.2, 3.3]], dtype=mstype.float32)
>>> output = GradNetWrtX(Net())(x, y)
>>> print(output)
[[1.4100001 1.5999999 6.6      ]
 [1.4100001 1.5999999 6.6      ]]
>>>
>>> class GradNetWrtXY(nn.Cell):
...     def __init__(self, net):
...         super(GradNetWrtXY, self).__init__()
...         self.net = net
...         self.grad_op = ops.GradOperation(get_all=True)
...     def construct(self, x, y):
...         gradient_function = self.grad_op(self.net)
...         return gradient_function(x, y)
>>>
>>> x = Tensor([[0.8, 0.6, 0.2], [1.8, 1.3, 1.1]], dtype=mstype.float32)
>>> y = Tensor([[0.1, 3.3, 1.1], [1.1, 0.2, 1.4], [1.1, 2.2, 0.3]], dtype=mstype.float32)
>>> output = GradNetWrtXY(Net())(x, y)
>>> print(output)
(Tensor(shape=[2, 3], dtype=Float32, value=
[[ 4.50000000e+00,  2.70000005e+00,  3.60000014e+00],
 [ 4.50000000e+00,  2.70000005e+00,  3.60000014e+00]]), Tensor(shape=[3, 3], dtype=Float32,␣
↪value=
[[ 2.59999990e+00,  2.59999990e+00,  2.59999990e+00],
 [ 1.89999998e+00,  1.89999998e+00,  1.89999998e+00],
 [ 1.30000007e+00,  1.30000007e+00,  1.30000007e+00]]))
>>>
>>> class GradNetWrtXYWithSensParam(nn.Cell):
```

(continues on next page)

(continued from previous page)

```
...     def __init__(self, net):
...         super(GradNetWrtXYWithSensParam, self).__init__()
...         self.net = net
...         self.grad_op = ops.GradOperation(get_all=True, sens_param=True)
...         self.grad_wrt_output = Tensor([[0.1, 0.6, 0.2], [0.8, 1.3, 1.1]], dtype=mstype.
↪float32)
...     def construct(self, x, y):
...         gradient_function = self.grad_op(self.net)
...         return gradient_function(x, y, self.grad_wrt_output)
>>>
>>> x = Tensor([[0.8, 0.6, 0.2], [1.8, 1.3, 1.1]], dtype=mstype.float32)
>>> y = Tensor([[0.11, 3.3, 1.1], [1.1, 0.2, 1.4], [1.1, 2.2, 0.3]], dtype=mstype.float32)
>>> output = GradNetWrtXYWithSensParam(Net())(x, y)
>>> print(output)
(Tensor(shape=[2, 3], dtype=Float32, value=
[[ 2.21099997e+00,  5.09999990e-01,  1.49000001e+00],
 [ 5.58800030e+00,  2.68000007e+00,  4.07000017e+00]]), Tensor(shape=[3, 3], dtype=Float32,␣
↪value=
[[ 1.51999998e+00,  2.81999993e+00,  2.14000010e+00],
 [ 1.09999990e+00,  2.04999995e+00,  1.54999995e+00],
 [ 9.00000036e-01,  1.54999995e+00,  1.25000000e+00]]))
>>>
>>> class GradNetWithWrtParams(nn.Cell):
...     def __init__(self, net):
...         super(GradNetWithWrtParams, self).__init__()
...         self.net = net
...         self.params = ParameterTuple(net.trainable_params())
...         self.grad_op = ops.GradOperation(get_by_list=True)
...     def construct(self, x, y):
...         gradient_function = self.grad_op(self.net, self.params)
...         return gradient_function(x, y)
>>>
>>> x = Tensor([[0.8, 0.6, 0.2], [1.8, 1.3, 1.1]], dtype=mstype.float32)
>>> y = Tensor([[0.11, 3.3, 1.1], [1.1, 0.2, 1.4], [1.1, 2.2, 0.3]], dtype=mstype.float32)
>>> output = GradNetWithWrtParams(Net())(x, y)
>>> print(output)
(Tensor(shape=[1], dtype=Float32, value= [ 2.15359993e+01]),)
>>>
>>> class GradNetWrtInputsAndParams(nn.Cell):
...     def __init__(self, net):
...         super(GradNetWrtInputsAndParams, self).__init__()
...         self.net = net
...         self.params = ParameterTuple(net.trainable_params())
...         self.grad_op = ops.GradOperation(get_all=True, get_by_list=True)
...     def construct(self, x, y):
...         gradient_function = self.grad_op(self.net, self.params)
...         return gradient_function(x, y)
>>>
>>> x = Tensor([[0.1, 0.6, 1.2], [0.5, 1.3, 0.1]], dtype=mstype.float32)
>>> y = Tensor([[0.12, 2.3, 1.1], [1.3, 0.2, 2.4], [0.1, 2.2, 0.3]], dtype=mstype.float32)
>>> output = GradNetWrtInputsAndParams(Net())(x, y)
>>> print(output)
((Tensor(shape=[2, 3], dtype=Float32, value=
[[ 3.51999998e+00,  3.90000010e+00,  2.59999990e+00],
 [ 3.51999998e+00,  3.90000010e+00,  2.59999990e+00]]), Tensor(shape=[3, 3], dtype=Float32,␣
↪value=
[[ 6.00000024e-01,  6.00000024e-01,  6.00000024e-01],
```

(continues on next page)

(continued from previous page)

```
 [ 1.89999998e+00,  1.89999998e+00,  1.89999998e+00],
 [ 1.30000007e+00,  1.30000007e+00,  1.30000007e+00]])), (Tensor(shape=[1], dtype=Float32,␣
↪value=
 [ 1.29020004e+01]),))
```

18.10.4 mindspore.ops.HookBackward

class mindspore.ops.**HookBackward**(*hook_fn*, *cell_id=''*)

This operation is used as a tag to hook gradient in intermediate variables. Note that this function is only supported in pynative mode.

Note: The hook function must be defined like *hook_fn(grad) -> new gradient or None*, where the 'grad' is the gradient passed to the primitive. The 'grad' may be modified by returning a new gradient and passed to next primitive. The difference between a hook function and callback of InsertGradientOf is that the hook function is executed in the python environment while callback will be parsed and added to the graph.

Parameters

- **hook_fn** (*Function*) – Python function. hook function.
- **cell_id** (*str, optional*) – Used to identify whether the function registered by the hook is actually registered on the specified cell object. For example, 'nn.Conv2d' is a cell object. Default: `""`, in this case, the system will automatically register a value of *cell_id*. The value of *cell_id* currently does not support custom values.

Inputs:

- **input** (Tensor) - The variable to hook.

Outputs:

- **output** (Tensor) - Returns *input* directly. *HookBackward* does not affect the forward result.

Raises

- **TypeError** – If *input* is not a tensor.
- **TypeError** – If *hook_fn* is not a function of python.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import mindspore as ms
>>> from mindspore import ops
>>> from mindspore import Tensor
>>> from mindspore.ops import GradOperation
>>> ms.set_context(mode=ms.PYNATIVE_MODE)
>>> def hook_fn(grad):
...     print(grad)
...
>>> hook = ops.HookBackward(hook_fn)
```

(continues on next page)

(continued from previous page)

```
>>> def hook_test(x, y):
...     z = x * y
...     z = hook(z)
...     z = z * y
...     return z
...
>>> grad_all = GradOperation(get_all=True)
>>> def backward(x, y):
...     return grad_all(hook_test)(x, y)
...
>>> output = backward(Tensor(1, ms.float32), Tensor(2, ms.float32))
(Tensor(shape=[], dtype=Float32, value= 2),)
>>> print(output)
(Tensor(shape=[], dtype=Float32, value= 4), Tensor(shape=[], dtype=Float32, value= 4))
```

18.10.5 mindspore.ops.HyperMap

class mindspore.ops.**HyperMap**(*ops=None*, *reverse=False*)

HyperMap will apply the set operation to input sequences.

Apply the operations to every element of the sequence or nested sequence. Different from *mindspore.ops.Map*, the *HyperMap* supports to apply on nested structure. The *HyperMap* also supports dynamic sequences as input, but it does not extend this support to nested dynamic sequences.

Parameters

- **ops** (*Union[MultitypeFuncGraph, None], optional*) – *ops* is the operation to apply. If *ops* is *None*, the operations should be put in the first input of the instance. Default is None.
- **reverse** (*bool, optional*) – The optimizer needs to be inverted in some scenarios to improve parallel performance, general users please ignore. *reverse* is the flag to decide if apply the operation reversely. Only supported in graph mode. Default is False.

Inputs:

- **args** (Tuple[sequence]) -
  - If *ops* is not *None*, all the inputs should be sequences with the same length. And each row of the sequences will be the inputs of the operation.
  - If *ops* is *None*, the first input is the operation, and the others are inputs.

Note: Except for the operation input, the number of inputs should be equal to the number of inputs to *ops*.

Outputs:

Sequence or nested sequence, the sequence of output after applying the function. e.g. *operation(args[0][i], args[1][i])*, *operation* is the function assigned by *ops*.

Raises

- **TypeError** – If *ops* is neither *mindspore.ops.MultitypeFuncGraph* nor None.
- **TypeError** – If *args* is not a Tuple.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> nest_tensor_list = ((Tensor(1, mstype.float32), Tensor(2, mstype.float32)),
...                     (Tensor(3, mstype.float32), Tensor(4, mstype.float32)))
>>> # square all the tensor in the nested list
>>>
>>> square = ops.MultitypeFuncGraph('square')
>>> @square.register("Tensor")
... def square_tensor(x):
...     return ops.square(x)
>>>
>>> common_map = ops.HyperMap()
>>> output = common_map(square, nest_tensor_list)
>>> print(output)
((Tensor(shape=[], dtype=Float32, value= 1), Tensor(shape=[], dtype=Float32, value= 4)),
(Tensor(shape=[], dtype=Float32, value= 9), Tensor(shape=[], dtype=Float32, value= 16)))
>>> square_map = ops.HyperMap(square, False)
>>> output = square_map(nest_tensor_list)
>>> print(output)
((Tensor(shape=[], dtype=Float32, value= 1), Tensor(shape=[], dtype=Float32, value= 4)),
(Tensor(shape=[], dtype=Float32, value= 9), Tensor(shape=[], dtype=Float32, value= 16)))
```

18.10.6 mindspore.ops.InsertGradientOf

class mindspore.ops.**InsertGradientOf**(*f*)

Attaches callback to the graph node that will be invoked on the node's gradient.

> **Warning:** In the callback, exercise caution when using side-effect operators, such as the TensorDump operator, as current support is incomplete.

Parameters

f (*Function*) – MindSpore's Function. Callback function.

Inputs:

- **input_x** (Any) - The graph node to attach to.

Outputs:

Tensor, returns *input_x* directly. *InsertGradientOf* does not affect the forward result.

Raises

TypeError – If *f* is not a function of MindSpore.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops, jit
>>> a = Tensor(np.array([1.0]).astype(np.float32))
>>> b = Tensor(np.array([0.2]).astype(np.float32))
>>> def clip_gradient(dx):
...     ret = dx
...     if ret > a:
...         ret = a
...
...     if ret < b:
...         ret = b
...
...     return ret
...
>>> clip = ops.InsertGradientOf(clip_gradient)
>>> grad_all = ops.GradOperation(get_all=True)
>>> def InsertGradientOfClipDemo():
...     def clip_test(x, y):
...         x = clip(x)
...         y = clip(y)
...         c = x * y
...         return c
...
...     @jit
...     def f(x, y):
...         return clip_test(x, y)
...
...     def fd(x, y):
...         return grad_all(clip_test)(x, y)
...
...     print("forward: ", f(Tensor(np.array([1.1]).astype(np.float32)),
...         Tensor(np.array([0.1]).astype(np.float32))))
...     print("clip_gradient:", fd(Tensor(np.array([1.1]).astype(np.float32)),
...         Tensor(np.array([0.1]).astype(np.float32))))
>>> InsertGradientOfClipDemo()
forward: [0.11000001]
clip_gradient: (Tensor(shape=[1], dtype=Float32, value= [ 2.00000003e-01]),
                Tensor(shape=[1], dtype=Float32, value= [ 1.00000000e+00]))
```

18.10.7 mindspore.ops.Morph

class mindspore.ops.**Morph**(*fn*, *infer_shape*, *infer_dtype*)

The *Morph* Primitive is used to encapsulate a user-defined function *fn*, allowing it to be used as a custom Primitive. The primary application scenario of the *Morph* Primitive is in the auto-parallel case after *GRAPH_MODE* mode, where collective communication operators are used within the user-defined *fn* to implement custom parallel computation logic, especially in scenarios where *fn* involves dynamic shapes. When the *Morph* Primitive is applied to inputs, it is actually the encapsulated user-defined function *fn* that is applied to the inputs. The main difference between the *Morph* Primitive and *mindspore.ops.Custom()* is that the former is expanded and replaced by the user-defined *fn* before automatic differentiation, so there is no need to implement a backward function.

Note:

- This primitive is only supported in GRAPH_MODE.

- *fn* must satisfy the syntax constraints of the graph mode.
- Users do not need to implement a custom backward function.
- *vararg*, *kwarg*, *kwonlyargs* and free variables are not supported in user-defined function.

Parameters

- **fn** (*Function*) –Mindspore's function, user-defined function.
- **infer_shape** (*Function*) –Mindspore's function, user-defined infer_shape function.
- **infer_dtype** (*Function*) –Mindspore's function, user-defined infer_dtype function.

Inputs:

The inputs of user-defined *fn*.

Outputs:

The outputs of user-defined *fn*.

Raises

RuntimeError –if not used in GRAPH_MODE.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import context, nn, ops, Tensor, Parameter
>>>
>>> np_weight0 = np.array([1.0, 2.0, 3.0])
>>> np_weight1 = np.array([4.0, 5.0, 6.0])
>>> np_input_x = np.array([7.0, 8.0, 9.0])
>>>
>>> def infer_dtype(args):
...     return args
>>>
>>> def infer_shape(args):
...     return args
>>>
>>> def mul_by(*args):
...     def inner(x):
...         return args[0] * x
...     return inner
>>>
>>> NUMBER_100 = 100
>>> class MorphNet(nn.Cell):
...     def __init__(self):
...         super(MorphNet, self).__init__()
...         self.weight0 = Parameter(Tensor(np_weight0, ms.float32), name="weight0")
...         self.weight1 = Parameter(Tensor(np_weight1, ms.float32), name="weight1")
...         self.mul_by_100 = ops.Morph(mul_by(NUMBER_100), infer_shape, infer_dtype)
...     def construct(self, x):
...         a = x * self.weight0
...         b = self.mul_by_100(a)
...         out = b * self.weight1
```

(continues on next page)

(continued from previous page)

```
...         return out
>>>
>>> context.set_context(mode=context.GRAPH_MODE)
>>> input_x = Tensor(np_input_x, ms.float32)
>>> net = MorphNet()
>>> grad_op = ops.GradOperation(get_all=True, get_by_list=True)
>>> grad_net = grad_op(net, net.trainable_params())
>>> bwd_out = grad_net(input_x)
>>> x_grad = bwd_out[0][0].asnumpy()
>>> weight0_grad = bwd_out[1][0].asnumpy()
>>> weight1_grad = bwd_out[1][1].asnumpy()
>>> print("x_grad", x_grad)
>>> print("weight0_grad", weight0_grad)
>>> print("weight1_grad", weight1_grad)
x_grad [ 400. 1000. 1800.]
weight0_grad [2800. 4000. 5400.]
weight1_grad [ 700. 1600. 2700.]
```

18.10.8 mindspore.ops.Map

class mindspore.ops.**Map**(*ops=None*, *reverse=False*)

Map will apply the set operation on input sequences.

Apply the operations to every element of the sequence.

Parameters

- **ops** (*Union[MultitypeFuncGraph, None], optional*) – *ops* is the operation to apply. If *ops* is *None*, the operations should be put in the first input of the instance. Default: None .
- **reverse** (*bool, optional*) – The optimizer needs to be inverted in some scenarios to improve parallel performance, general users please ignore. *Reverse* is the flag to decide if apply the operation reversely. Only supported in graph mode. Default is False .

Inputs:

- **args** (Tuple[sequence]) - If *ops* is not *None*, all the inputs should be the same length sequences, and each row of the sequences. e.g. If the length of args is 2, and for *i* in length of each sequence *(args[0][i], args[1][i])* will be the input of the operation.

  If *ops* is *None*, the first input is the operation, and the other is the sequence.

Outputs:

Sequence, the sequence of output after applying the ops function. e.g. *ops(args[0][i], args[1][i])*.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import dtype as mstype
>>> from mindspore import Tensor, ops
>>> from mindspore.ops import MultitypeFuncGraph, Map
>>> tensor_list = (Tensor(1, mstype.float32), Tensor(2, mstype.float32), Tensor(3, mstype.
↪float32))
>>> # square all the tensor in the list
>>>
>>> square = MultitypeFuncGraph('square')
>>> @square.register("Tensor")
... def square_tensor(x):
...     return ops.square(x)
>>>
>>> common_map = Map()
>>> output = common_map(square, tensor_list)
>>> print(output)
(Tensor(shape=[], dtype=Float32, value= 1), Tensor(shape=[], dtype=Float32, value= 4),
Tensor(shape=[], dtype=Float32, value= 9))
>>> square_map = Map(square, False)
>>> output = square_map(tensor_list)
>>> print(output)
(Tensor(shape=[], dtype=Float32, value= 1), Tensor(shape=[], dtype=Float32, value= 4),
Tensor(shape=[], dtype=Float32, value= 9))
```

18.10.9 mindspore.ops.MultitypeFuncGraph

class mindspore.ops.**MultitypeFuncGraph**(*name*, *read_value=False*)

MultitypeFuncGraph is a class used to generate overloaded functions, considering different types as inputs. Initialize an *MultitypeFuncGraph* object with name, and use *register* with input types as the decorator for the function to be registered. And the object can be called with different types of inputs, and work with *HyperMap* and *Map*.

Parameters

- **name** (*str*) – Operator name.
- **read_value** (*bool, optional*) – If the registered function do not need to set value on Parameter, and all inputs will pass by value, set *read_value* to `True` . Default: `False` .

Raises

ValueError – If failed to find a matching function for the given arguments.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> # `add` is a metagraph object which will add two objects according to
>>> # input type using ".register" decorator.
>>> from mindspore import Tensor
>>> from mindspore import dtype as mstype
>>> from mindspore import ops
>>>
>>> tensor_add = ops.Add()
>>> add = ops.MultitypeFuncGraph('add')
>>> @add.register("Number", "Number")
... def add_scala(x, y):
...     return x + y
>>> @add.register("Tensor", "Tensor")
... def add_tensor(x, y):
...     return tensor_add(x, y)
>>> output = add(1, 2)
>>> print(output)
3
>>> output = add(Tensor([0.1, 0.6, 1.2], dtype=mstype.float32), Tensor([0.1, 0.6, 1.2],␣
↪dtype=mstype.float32))
>>> print(output)
[0.2 1.2 2.4]
```

register(**type_names*)

Register a function for the given type string.

Parameters

type_names (Union[str, *mindspore.dtype*]) –Inputs type names or types list.

Returns

decorator, a decorator to register the function to run, when called under the types described in *type_names*.

18.10.10 mindspore.ops.Partial

class mindspore.ops.**Partial**

Makes a partial function instance. Partial function can be used to derived specialized functions from general functions by fixing the value of certain number of arguments.

Inputs:

- **args** (Union[FunctionType, Tensor]) - The function and bind arguments.

Outputs:

FunctionType, partial function bound with arguments.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import Tensor
>>> from mindspore import ops
>>> def show_input(x, y, z):
...     return x, y, z
>>> partial = ops.Partial()
>>> partial_show_input = partial(show_input, Tensor(1))
>>> output1 = partial_show_input(Tensor(2), Tensor(3))
>>> print(output1)
(Tensor(shape=[], dtype=Int64, value= 1), Tensor(shape=[], dtype=Int64, value= 2),␣
↪Tensor(shape=[], dtype=Int64,
 value= 3))
>>> output2 = partial_show_input(Tensor(3), Tensor(4))
>>> print(output2)
(Tensor(shape=[], dtype=Int64, value= 1), Tensor(shape=[], dtype=Int64, value= 3),␣
↪Tensor(shape=[], dtype=Int64,
 value= 4))
```

18.10.11 mindspore.ops.Scan

class mindspore.ops.**Scan**

Scan a function over an array while the processing of the current element depends on the execution result of the previous element. The execution logic of the Scan operator can be roughly represented by the following code:

```
def Scan(loop_func, init, xs, length=None):
    if xs is None:
        xs = [None] * length
    carry = init
    ys = []
    for x in xs:
        carry, y = loop_func(carry, x)
        ys.append(y)
    return carry, ys
```

The current Scan operator has the following syntactic limitations:

- Using a side-effect function as *loop_func* is currently not support, such as operations that modify parameters, global variables, etc.
- The first element of the return value of *loop_func* being of a different type or shape from the *init_val* is currently not support.

> **Warning:** This is an experimental API that is subject to change or deletion.

Inputs:

- **loop_func** (Function) - The loop function.
- **init** (Union[Tensor, number, str, bool, list, tuple, dict]) - An initial loop carry value.
- **xs** (Union[tuple, list, None]) - The value over which to scan.
- **length** (Union[int, None], optional) - The size of xs. Default: `None` .
- **unroll** (bool, optional) - The flag for whether to perform loop unrolling in compile process. Default: `True` .

Outputs:

Tuple(Union[Tensor, number, str, bool, list, tuple, dict], list). Output of scan loop, a tuple with two elements, the first element has same type and shape with init argument, and the second is a list containing the results of each loop.

Raises

- **TypeError** – If *loop_func* is not a function.
- **TypeError** – If *xs* is not a tuple, a list or None.
- **TypeError** – If *length* is not an int or None.
- **TypeError** – If *unroll* is not a bool.
- **ValueError** – If *loop_func* cannot take *init* and element of *xs* as inputs.
- **ValueError** – If the return value of *loop_func* is not a tuple with two elements, or the first element of the tuple has different type or shape from *init* .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> def cumsum(res, el):
...     res = res + el
...     return res, res
...
>>> a = [1, 2, 3, 4]
>>> result_init = 0
>>> scan_op = ops.Scan()
>>> result = scan_op(cumsum, result_init, a)
>>> print(result == (10, [1, 3, 6, 10]))
True
```

18.10.12 mindspore.ops.WhileLoop

class mindspore.ops.**WhileLoop**

Provide a useful op for reducing compilation times of while loop. The execution logic of the WhileLoop operator can be roughly represented by the following code:

```
def WhileLoop(cond_func, loop_func, init_val):
    while(cond_func(init_val)):
        init_val = loop_func(init_val)
    return init_val
```

The current WhileLoop operator has the following syntactic limitations:

- Using a side-effect function as *loop_func* is currently not support, such as operations that modify parameters, global variables, etc.
- The return value of *loop_func* being of a different type or shape from the *init_val* is currently not support.

Warning: This is an experimental API that is subject to change or deletion.

Inputs:

- **cond_func** (Function) - The condition function.
- **loop_func** (Function) - The loop function, take one argument and return value has the same type with input argument.
- **init_val** (Union[Tensor, number, str, bool, list, tuple, dict]) - The initial value.

Outputs:

Union[Tensor, number, str, bool, list, tuple, dict], the final result of the while loop, has same type and shape with input *init_val* .

Raises

- **TypeError** –If *cond_func* is not a function.
- **TypeError** –If *loop_func* is not a function.
- **ValueError** –If *loop_func* cannot take *init_val* as input or has different output type or shape with *init_val* .

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore import ops
>>> def loop_while_fun(init_val):
...     val = init_val
...     val = val + 1
...     return val
...
>>> init_state = 10
>>> while_loop = ops.WhileLoop()
>>> result = while_loop(lambda x : x < 100, loop_while_fun, init_state)
>>> print(result)
100
```

18.11 Operator Information Registration

mindspore.ops.AiCPURegOp	Class for AiCPU operator information registration.
mindspore.ops.CustomRegOp	Class used for generating the registration information for the *func* parameter of *mindspore.ops.Custom*.
mindspore.ops.DataType	Various combinations of dtype and format of Ascend ops.
mindspore.ops.TBERegOp	Class for TBE operator information registration.
mindspore.ops.get_vm_impl_fn	Gets the virtual implementation function by a primitive object or primitive name.

18.11.1 mindspore.ops.AiCPURegOp

class mindspore.ops.**AiCPURegOp**(*op_name*)

Class for AiCPU operator information registration.

Parameters

op_name (*str*) – Name of operator.

Examples

```
>>> from mindspore.ops import AiCPURegOp, DataType
>>> stack_op_info = AiCPURegOp("Stack") \
...     .fusion_type("OPAQUE") \
...     .attr("axis", "int") \
...     .input(0, "x", "dynamic") \
...     .output(0, "y", "required") \
...     .dtype_format(DataType.I8_Default, DataType.I8_Default) \
...     .dtype_format(DataType.I16_Default, DataType.I16_Default) \
...     .dtype_format(DataType.I32_Default, DataType.I32_Default) \
...     .dtype_format(DataType.I64_Default, DataType.I64_Default) \
...     .dtype_format(DataType.U8_Default, DataType.U8_Default) \
...     .dtype_format(DataType.U16_Default, DataType.U16_Default) \
...     .dtype_format(DataType.U32_Default, DataType.U32_Default) \
...     .dtype_format(DataType.U64_Default, DataType.U64_Default) \
...     .dtype_format(DataType.F16_Default, DataType.F16_Default) \
...     .dtype_format(DataType.F32_Default, DataType.F32_Default) \
...     .dtype_format(DataType.F64_Default, DataType.F64_Default) \
...     .dtype_format(DataType.BOOL_Default, DataType.BOOL_Default) \
...     .get_op_info()
>>>
```

18.11.2 mindspore.ops.CustomRegOp

class mindspore.ops.**CustomRegOp**(*op_name='Custom'*)

Class used for generating the registration information for the *func* parameter of *mindspore.ops.Custom*. The registration information mainly specifies the supported data types and formats of input and output tensors, attributes and target of *func*.

Parameters

op_name (*str, optional*) – kernel name. The name will be record in the reg_op_name attr of the kernel node. Besides, the operator will generate a unique name automatically to identify the reg info. Default: "Custom" .

Examples

```
>>> from mindspore.ops import CustomRegOp, DataType
>>> custom_op_ascend_info = CustomRegOp() \
...     .input(0, "x", "dynamic") \
...     .output(0, "y") \
...     .dtype_format(DataType.F16_Default, DataType.F16_Default) \
...     .dtype_format(DataType.F32_Default, DataType.F32_Default) \
...     .target("Ascend") \
...     .get_op_info()
```

attr(*name=None*, *param_type=None*, *value_type=None*, *default_value=None*, ***kwargs*)

Specifies the attributes information for the *func* parameter of *mindspore.ops.Custom*. Each invocation of this function will generate one attribute information, that means, if *func* has two attributes, then this function should be invoked two times continuously. The attributes information will be generated as a dict: {"name": *name*, "param_type": *param_type*, "value_type": *value_type*, "default_value": *default_value*}.

Parameters

- **name** (*str*) −Name of the attribute. If `None` , key "name" will not appear in the attributes tensor information dict. Default: `None` .
- **param_type** (*str*) −Parameter type of the attribute, can be one of ["required", "optional"]. If `None` , key "param_type" will not appear in the attributes tensor information dict. Default: `None` .
  - "required": means must provide a value for this attribute either by setting a default value in the registration information or providing an input value when calling the Custom operator.
  - "optional": means does not have to provide a value for this attribute.
- **value_type** (*str*) −Value type of the attribute, can be one of ["int", "str", "bool", "float", "listInt", "listStr", "listBool", "listFloat"]. If `None` , key "value_type" will not appear in the attributes tensor information dict. Default: `None` .
  - "int": string representation of Python type int.
  - "str": string representation of Python type str.
  - "bool": string representation of Python type bool.
  - "float": string representation of Python type float.
  - "listInt": string representation of Python type list of int.
  - "listStr": string representation of Python type list of str.
  - "listBool": string representation of Python type list of bool.
  - "listFloat": string representation of Python type list of float.
- **default_value** (*str*) −Default value of the attribute. *default_value* and *value_type* are used together. If the real default value of the attribute is float type with value 1.0, then the *value_type* should be "float" and *default_value* should be "1.0". If the real default value of the attribute is a list of int with value [1, 2, 3], then the *value_type* should be "listInt" and *default_value* should be "1,2,3", each item should split by ','. If `None` , means the attribute has no default value and key "default_value" will not appear in the attributes tensor information dict. It is used for "akg", "aicpu" and "tbe" Custom operators currently. Default: `None` .
- **kwargs** (*dict*) −Other information of the attribute, used for extension.

Raises

- **TypeError** −If *name* is neither str nor None.
- **TypeError** −If *param_type* is neither str nor None.
- **TypeError** −If *value_type* is neither str nor None.
- **TypeError** −If *default_value* is neither str nor None.

dtype_format(**args*)

Specifies the supported data type and format of each input tensor and output tensor for the *func* parameter of *mindspore.ops.Custom*. This function should be invoked after *input* and *output* function as shown in the above example.

Parameters

args (*tuple*) −A tuple of (data type, format) pair, the length of *args* should be equal to the sum of input tensors and output tensors. Each item in *args* is also a tuple, tuple[0] and tuple[1] are both str type, which specifies the data type and format of a tensor respectively. *mindspore.ops.DataType* provides many predefined (data type, format) combinations, for example, *DataType.F16_Default* means the data type is float16 and the format is default format.

Raises

ValueError −If the size of *args* not equal to the sum of input tensors and output tensors.

get_op_info()

Return the generated registration information as a dict. This function should be invoked at last on the *CustomRegOp* instance as shown in the above example.

input(*index=None*, *name=None*, *param_type='required'*, ***kwargs*)

Specifies the input tensor information for the *func* parameter of *mindspore.ops.Custom*. Each invocation of this function will generate one input tensor information, that means, if *func* has two input tensors, then this function should be invoked two times continuously. The input tensor information will be generated as a dict: {"index": *index*, "name": *name*, "param_type": *param_type*}.

Parameters

- **index** (*int*) −Index of the input, starts from 0. 0 means the first input tensor, 1 means the second input tensor and so on. If `None` , key "index" will not appear in the input tensor information dict. Default: `None` .
- **name** (*str*) −Name of the *index* 'th input. If `None` , key "name" will not appear in the input tensor information dict. Default: `None` .
- **param_type** (*str*) −Parameter type of the *index* 'th input, can be one of [`"required"` , `"dynamic"` , `"optional"`]. If `None` , key "param_type" will not appear in the input tensor information dict. Default: `"required"` .
  - `"required"`: means the *index* 'th input exist and can only be a single tensor.
  - `"dynamic":` means the *index* 'th input exist and may be multiple tensors, such as the input of AddN.
  - `"optional"`: means the *index* 'th input may exist and be a single tensor or may not exist.
- **kwargs** (*dict*) −Other information of the input, used for extension.

Raises

- **TypeError** −If *index* is neither int nor None.
- **TypeError** −If *name* is neither str nor None.
- **TypeError** −If *param_type* is neither str nor None.

output(*index=None*, *name=None*, *param_type='required'*, ***kwargs*)

Specifies the output tensor information for the *func* parameter of *mindspore.ops.Custom*. Each invocation of this function will generate one output tensor information, which means, if *func* has two output tensors, then this function should be invoked two times continuously. The output tensor information will be generated as a dict: {"index": *index*, "name": *name*, "param_type": *param_type*}.

Parameters

- **index** (*int*) −Index of the output, starts from 0. 0 means the first output tensor, 1 means the second output tensor and so on. If `None` , key "index" will not appear in the output tensor information dict. Default: `None` .
- **name** (*str*) −Name of the *index* 'th output. If `None` , key "name" will not appear in the output tensor information dict. Default: `None` .
- **param_type** (*str*) −Parameter type of the *index* 'th output, can be one of [`"required"` , `"dynamic"` , `"optional"`]. If `None` , key "param_type" will not appear in the output tensor information dict. Default: `"required"` .

- `"required"`: means the *index* 'th output exist and can only be a single tensor.
- `"dynamic"`: means the *index* 'th output exist and may be multiple tensors.
- `"optional"`: means the *index* 'th output may exist and be a single tensor or may not exist.

- **kwargs** (*dict*) –Other information of the output, used for extension.

Raises

- **TypeError** –If *index* is neither int nor None.
- **TypeError** –If *name* is neither str nor None.
- **TypeError** –If *param_type* is neither str nor None.

target(*target=None*)

Specifies the target that this registration information is used for.

Parameters

target (*str*) –Device target for current operator information, should be one of ["Ascend", "GPU", "CPU"]. For the same *func* of *mindspore.ops.Custom*, it may support different data types and formats on different targets, use *target* to specify which target that this registration information is used for. If `None` , it will be inferred automatically inside *mindspore.ops.Custom*. Default: `None` .

Raises

TypeError –If *target* is neither str nor None.

18.11.3 mindspore.ops.DataType

class mindspore.ops.**DataType**

Various combinations of dtype and format of Ascend ops.

current support:

```
None_None = ("", "")
None_Default = ("", "DefaultFormat")
BOOL_None = ("bool", "")
BOOL_Default = ("bool", "DefaultFormat")
BOOL_5HD = ("bool", "NC1HWC0")
BOOL_FracZ = ("bool", "FRACTAL_Z")
BOOL_FracNZ = ("bool", "FRACTAL_NZ")
BOOL_C1HWNCoC0 = ("bool", "C1HWNCoC0")
BOOL_NCHW = ("bool", "NCHW")
BOOL_NHWC = ("bool", "NHWC")
BOOL_HWCN = ("bool", "HWCN")
BOOL_NDHWC = ("bool", "NDHWC")
BOOL_ChannelLast = ("bool", "ChannelLast")

I8_None = ("int8", "")
I8_Default = ("int8", "DefaultFormat")
I8_5HD = ("int8", "NC1HWC0")
I8_FracZ = ("int8", "FRACTAL_Z")
I8_FracNZ = ("int8", "FRACTAL_NZ")
I8_C1HWNCoC0 = ("int8", "C1HWNCoC0")
I8_NCHW = ("int8", "NCHW")
I8_NHWC = ("int8", "NHWC")
I8_HWCN = ("int8", "HWCN")
I8_NDHWC = ("int8", "NDHWC")
```

(continues on next page)

(continued from previous page)

```
I8_ChannelLast = ("int8", "ChannelLast")
I8_NDC1HWC0 = ("int8", "NDC1HWC0")

U8_None = ("uint8", "")
U8_Default = ("uint8", "DefaultFormat")
U8_5HD = ("uint8", "NC1HWC0")
U8_FracZ = ("uint8", "FRACTAL_Z")
U8_FracNZ = ("uint8", "FRACTAL_NZ")
U8_C1HWNCoC0 = ("uint8", "C1HWNCoC0")
U8_NCHW = ("uint8", "NCHW")
U8_NHWC = ("uint8", "NHWC")
U8_HWCN = ("uint8", "HWCN")
U8_NDHWC = ("uint8", "NDHWC")
U8_ChannelLast = ("uint8", "ChannelLast")
U8_NDC1HWC0 = ("uint8", "NDC1HWC0")

I16_None = ("int16", "")
I16_Default = ("int16", "DefaultFormat")
I16_5HD = ("int16", "NC1HWC0")
I16_FracZ = ("int16", "FRACTAL_Z")
I16_FracNZ = ("int16", "FRACTAL_NZ")
I16_C1HWNCoC0 = ("int16", "C1HWNCoC0")
I16_NCHW = ("int16", "NCHW")
I16_NHWC = ("int16", "NHWC")
I16_HWCN = ("int16", "HWCN")
I16_NDHWC = ("int16", "NDHWC")
I16_ChannelLast = ("int16", "ChannelLast")

U16_None = ("uint16", "")
U16_Default = ("uint16", "DefaultFormat")
U16_5HD = ("uint16", "NC1HWC0")
U16_FracZ = ("uint16", "FRACTAL_Z")
U16_FracNZ = ("uint16", "FRACTAL_NZ")
U16_C1HWNCoC0 = ("uint16", "C1HWNCoC0")
U16_NCHW = ("uint16", "NCHW")
U16_NHWC = ("uint16", "NHWC")
U16_HWCN = ("uint16", "HWCN")
U16_NDHWC = ("uint16", "NDHWC")
U16_ChannelLast = ("uint16", "ChannelLast")

I32_None = ("int32", "")
I32_Default = ("int32", "DefaultFormat")
I32_5HD = ("int32", "NC1HWC0")
I32_FracZ = ("int32", "FRACTAL_Z")
I32_FracNZ = ("int32", "FRACTAL_NZ")
I32_C1HWNCoC0 = ("int32", "C1HWNCoC0")
I32_NCHW = ("int32", "NCHW")
I32_NHWC = ("int32", "NHWC")
I32_HWCN = ("int32", "HWCN")
I32_NDHWC = ("int32", "NDHWC")
I32_ChannelLast = ("int32", "ChannelLast")

U32_None = ("uint32", "")
U32_Default = ("uint32", "DefaultFormat")
U32_5HD = ("uint32", "NC1HWC0")
U32_FracZ = ("uint32", "FRACTAL_Z")
U32_FracNZ = ("uint32", "FRACTAL_NZ")
```

(continues on next page)

(continued from previous page)

```
U32_C1HWNCoC0 = ("uint32", "C1HWNCoC0")
U32_NCHW = ("uint32", "NCHW")
U32_NHWC = ("uint32", "NHWC")
U32_HWCN = ("uint32", "HWCN")
U32_NDHWC = ("uint32", "NDHWC")
U32_ChannelLast = ("uint32", "ChannelLast")

I64_None = ("int64", "")
I64_Default = ("int64", "DefaultFormat")
I64_5HD = ("int64", "NC1HWC0")
I64_FracZ = ("int64", "FRACTAL_Z")
I64_FracNZ = ("int64", "FRACTAL_NZ")
I64_C1HWNCoC0 = ("int64", "C1HWNCoC0")
I64_NCHW = ("int64", "NCHW")
I64_NHWC = ("int64", "NHWC")
I64_HWCN = ("int64", "HWCN")
I64_NDHWC = ("int64", "NDHWC")
I64_ChannelLast = ("int64", "ChannelLast")

U64_None = ("uint64", "")
U64_Default = ("uint64", "DefaultFormat")
U64_5HD = ("uint64", "NC1HWC0")
U64_FracZ = ("uint64", "FRACTAL_Z")
U64_FracNZ = ("uint64", "FRACTAL_NZ")
U64_C1HWNCoC0 = ("uint64", "C1HWNCoC0")
U64_NCHW = ("uint64", "NCHW")
U64_NHWC = ("uint64", "NHWC")
U64_HWCN = ("uint64", "HWCN")
U64_NDHWC = ("uint64", "NDHWC")
U64_ChannelLast = ("uint64", "ChannelLast")

F16_None = ("float16", "")
F16_Default = ("float16", "DefaultFormat")
F16_5HD = ("float16", "NC1HWC0")
F16_FracZ = ("float16", "FRACTAL_Z")
F16_FracNZ = ("float16", "FRACTAL_NZ")
F16_C1HWNCoC0 = ("float16", "C1HWNCoC0")
F16_NCHW = ("float16", "NCHW")
F16_NHWC = ("float16", "NHWC")
F16_HWCN = ("float16", "HWCN")
F16_NDHWC = ("float16", "NDHWC")
F16_NCDHW = ("float16", "NCDHW")
F16_DHWCN = ("float16", "DHWCN")
F16_NDC1HWC0 = ("float16", "NDC1HWC0")
F16_FRACTAL_Z_3D = ("float16", "FRACTAL_Z_3D")
F16_FracZNLSTM = ("float16", "FRACTAL_ZN_LSTM")
F16_FracZNRNN = ("float16", "FRACTAL_ZN_RNN")
F16_ND_RNNBIAS = ("float16", "ND_RNN_BIAS")
F16_ChannelLast = ("float16", "ChannelLast")

F32_None = ("float32", "")
F32_Default = ("float32", "DefaultFormat")
F32_5HD = ("float32", "NC1HWC0")
F32_FracZ = ("float32", "FRACTAL_Z")
F32_FracNZ = ("float32", "FRACTAL_NZ")
F32_C1HWNCoC0 = ("float32", "C1HWNCoC0")
F32_NCHW = ("float32", "NCHW")
```

(continues on next page)

(continued from previous page)

```
F32_NHWC = ("float32", "NHWC")
F32_HWCN = ("float32", "HWCN")
F32_NDHWC = ("float32", "NDHWC")
F32_NCDHW = ("float32", "NCDHW")
F32_DHWCN = ("float32", "DHWCN")
F32_NDC1HWC0 = ("float32", "NDC1HWC0")
F32_FRACTAL_Z_3D = ("float32", "FRACTAL_Z_3D")
F32_FracZNLSTM = ("float32", "FRACTAL_ZN_LSTM")
F32_FracZNRNN = ("float32", "FRACTAL_ZN_RNN")
F32_ND_RNNBIAS = ("float32", "ND_RNN_BIAS")
F32_ChannelLast = ("float32", "ChannelLast")

F64_None = ("float64", "")
F64_Default = ("float64", "DefaultFormat")
F64_5HD = ("float64", "NC1HWC0")
F64_FracZ = ("float64", "FRACTAL_Z")
F64_FracNZ = ("float64", "FRACTAL_NZ")
F64_C1HWNCoC0 = ("float64", "C1HWNCoC0")
F64_NCHW = ("float64", "NCHW")
F64_NHWC = ("float64", "NHWC")
F64_HWCN = ("float64", "HWCN")
F64_NDHWC = ("float64", "NDHWC")
F64_ChannelLast = ("float64", "ChannelLast")

C64_Default = ("complex64", "DefaultFormat")
C128_Default = ("complex128", "DefaultFormat")
```

18.11.4 mindspore.ops.TBERegOp

class mindspore.ops.**TBERegOp**(*op_name*)

Class for TBE operator information registration. TBE (Tensor Boost Engine) is the Ascend operator development tool, which is extended on the basis of the TVM framework to develop custom operators.

Parameters

op_name (*str*) – Name of operator.

Examples

```
>>> from mindspore.ops import TBERegOp, DataType
>>> op_name_op_info = TBERegOp("OpName") \
...     .fusion_type("ELEMWISE") \
...     .async_flag(False) \
...     .binfile_name("op_name.so") \
...     .compute_cost(10) \
...     .kernel_name("op_name") \
...     .partial_flag(True) \
...     .op_pattern("formatAgnostic") \
...     .need_check_supported(True) \
...     .dynamic_shape(True) \
...     .dynamic_rank_support(True) \
...     .dynamic_compile_static(True) \
...     .attr("format", "required", "str", "all") \
...     .input(0, "x1", None, "required", None) \
```

(continues on next page)

(continued from previous page)

```
...     .input(0, "x2", None, "required", None) \
...     .input(1, "axis", None, "required", None) \
...     .output(0, "y", True, "required", "all") \
...     .real_input_index([1, 0]) \
...     .input_to_attr_index([2]) \
...     .unknown_shape_formats(["ND", "ND", "ND", "ND"]) \
...     .reshape_type("NC") \
...     .is_dynamic_format(True) \
...     .dtype_format(DataType.F16_None, DataType.F16_None, DataType.F16_None, DataType.F16_
↪None) \
...     .dtype_format(DataType.F32_None, DataType.F32_None, DataType.F32_None, DataType.F32_
↪None) \
...     .dtype_format(DataType.I32_None, DataType.I32_None, DataType.I32_None, DataType.I32_
↪None) \
...     .get_op_info()
>>>
```

async_flag(*async_flag=False*)
Define the calculation efficiency of the operator, whether the asynchronous calculation is supported.

Parameters

async_flag (*bool, optional*) – Value of async flag. Default: `False` .

attr(*name=None*, *param_type=None*, *value_type=None*, *value=None*, *default_value=None*, ***kwargs*)
Register TBE op attribute information.

Parameters

- **name** (*str, optional*) – Name of the attribute. Default: `None` .
- **param_type** (*str, optional*) – Param type of the attribute. Default: `None` .
- **value_type** (*str, optional*) – Type of the attribute. Default: `None` .
- **value** (*str, optional*) – Value of the attribute. Default: `None` .
- **default_value** (*str, optional*) – Default value of attribute. Default: `None` .
- **kwargs** (*dict*) – Other information of the attribute.

binfile_name(*binfile_name*)
Set the binary file name of the operator, it is optional.

Parameters

binfile_name (*str*) – The binary file name of the operator.

compute_cost(*compute_cost=10*)
Define the calculation efficiency of operator, which refers to the value of the cost model in the tiling module.

Parameters

compute_cost (*int, optional*) – Value of compute cost. Default: `10` .

dynamic_compile_static(*dynamic_compile_static=False*)
Whether the operator supports dynamic compile static.

Parameters

dynamic_compile_static (*bool, optional*) – An identifier that indicates whether the operator supports dynamic compile static. Default: `False` .

dynamic_rank_support (*dynamic_rank_support*)
Description whether the operator supports dynamic rank (dynamic dimension).

Parameters

dynamic_rank_support (*bool, optional*) –Description whether the operator supports dynamic rank (dynamic dimension). True: indicates that dynamic rank is supported. False: indicates that the operator does not support dynamic rank. Default: `False` .

dynamic_shape (*dynamic_shape=False*)
Whether the operator supports dynamic shape.

Parameters

dynamic_shape (*bool, optional*) –Value of dynamic shape. Default: `False` .

input (*index=None*, *name=None*, *need_compile=None*, *param_type=None*, *shape=None*, *value_depend=None*, ***kwargs*)
Register TBE op input information.

Parameters

- **index** (*int, optional*) –Order of the input. Default: `None` .
- **name** (*str, optional*) –Name of the input. Default: `None` .
- **need_compile** (*bool, optional*) –Whether the input needs to be compiled or not. Default: `None` .
- **param_type** (*str, optional*) –Type of the input. Default: `None` .
- **shape** (*str, optional*) –Shape of the input. Default: `None` .
- **value_depend** (*str, optional*) –Whether the input is constant value depend. Default: `None` .
- **kwargs** (*dict*) –Other information of the input.

input_to_attr_index (*input_to_attr_index*)
Description the index of input need to cast to attr.

Parameters

input_to_attr_index (*list*) –Value of input_to_attr_index. Default: `()` .

is_dynamic_format (*is_dynamic_format=False*)
Whether the operator needs calop_select_format api.

Parameters

is_dynamic_format (*bool, optional*) –Value of is_dynamic_format. Default: `False` .

kernel_name (*kernel_name*)
The name of operator kernel.

Parameters

kernel_name (*str*) –Name of operator kernel.

need_check_supported (*need_check_supported=False*)
Whether the operator needs check supports.

Parameters

need_check_supported (*bool, optional*) –Value of need_check_supported. Default: `False` .

op_pattern (*pattern=None*)
The behavior type of operator, such as broadcast, reduce and so on.

Parameters

pattern (*str, optional*) –Value of op pattern, e.g. "broadcast", "reduce". Default: `None` .

output (*index=None*, *name=None*, *need_compile=None*, *param_type=None*, *shape=None*, ***kwargs*)
Register TBE op output information.

Parameters

- **index** (*int, optional*) –Order of the output. Default: `None` .
- **name** (*str, optional*) –Name of the output. Default: `None` .
- **need_compile** (*bool, optional*) –Whether the output needs to be compiled or not. Default: `None` .
- **param_type** (*str, optional*) –Type of the output. Default: `None` .
- **shape** (*str, optional*) –Shape of the output. Default: `None` .
- **kwargs** (*dict*) –Other information of the output.

partial_flag (*partial_flag=True*)
Define the calculation efficiency of operator, whether the partial calculation is supported.

Parameters

partial_flag (*bool, optional*) –Value of partial flag. Default: `True` .

real_input_index (*real_input_index*)
Description operator front end and tbe operator input mapping.

Parameters

real_input_index (*list*) –Value of real_input_index. Default: `()` .

reshape_type (*reshape_type*)
Reshape type of operator.

Parameters

reshape_type (*str*) –Value of reshape type. For example, if the input shape is $(2, 3)$ and *reshape_type* is set to "CH", then the new shape is $(1, 2, 3, 1)$. "CH" means the C and H dimensions are kept and new dimensions are added for N and W dimension.

unknown_shape_formats (*unknown_shape_formats*)
Description data arrangement of operator input / output tensor in dynamic shape scene.

Parameters

unknown_shape_formats (*list*) –Description data arrangement of operator input / output tensor in dynamic shape scene.

18.11.5 mindspore.ops.get_vm_impl_fn

mindspore.ops.**get_vm_impl_fn** (*prim*)
Gets the virtual implementation function by a primitive object or primitive name.

Parameters

prim (*Union[Primitive, str]*) –primitive object or name for operator register.

Note: This mechanism applied for debugging currently.

Returns

function, vm function

Examples

```
>>> from mindspore.ops import vm_impl_registry, get_vm_impl_fn
...
>>> @vm_impl_registry.register("Type")
... def vm_impl_dtype(self):
...   def vm_impl(x):
...     return type(x)
...   return vm_impl
...
>>> fn = get_vm_impl_fn("Type")
>>> out = fn(1.0)
>>> print(out)
<class 'float'>
```

18.12 Customizing Operator

API Name	Description	Supported Platforms
mindspore.ops.Custom	*Custom* primitive is used for user defined operators and is to enhance the expressive ability of built-in primitives.	Ascend GPU CPU
mindspore.ops.CustomOpBuilder	CustomOpBuilder is used to initialize and configure custom operators for MindSpore.	Ascend CPU
mindspore.ops.custom_info_register	A decorator which is used to bind the registration information to the *func* parameter of *mindspore.ops.Custom*.	Ascend GPU CPU
mindspore.ops.kernel	The decorator of the Hybrid DSL function for the Custom Op.	GPU CPU

18.12.1 mindspore.ops.Custom

class mindspore.ops.**Custom**(*func*, *out_shape=None*, *out_dtype=None*, *func_type='hybrid'*, *bprop=None*, *reg_info=None*)

Custom primitive is used for user defined operators and is to enhance the expressive ability of built-in primitives. You can construct a *Custom* object with a predefined function, which describes the computation logic of a user defined operator. You can also construct another *Custom* object with another predefined function if needed. Then these *Custom* objects can be directly used in neural networks. Detailed description and introduction of user-defined operators, including correct writing of parameters, please refer to Custom Operators Tutorial .

Warning:

- This is an experimental API that is subject to change.

Note: The supported platforms are determined by the input *func_type*. The supported platforms are as follows:

- "aot": supports ["GPU", "CPU", "Ascend"].

- "pyfunc": supports ["CPU"].
- "julia": supports ["CPU"].

Parameters

- **func** (*Union[function, str]*) –
  - function: If func is of function type, then func should be a Python function which describes the computation logic of a user defined operator.
  - str: If func is of str type, then str should be a path of file along with a function name. This could be used when func_type is "aot" or "julia".

    1. for "aot":

       a) GPU/CPU platform. "aot" means ahead of time, in which case Custom directly launches user defined "xxx.so" file as an operator. Users need to compile a handwriting "xxx.cu"/"xxx.cc" file into "xxx.so" ahead of time, and offer the path of the file along with a function name.

       * "xxx.so" file generation:

         1) GPU Platform: Given user defined "xxx.cu" file (ex. "{path}/add.cu"), use nvcc command to compile it.(ex. "nvcc –shared -Xcompiler -fPIC -o add.so add.cu")

         2) CPU Platform: Given user defined "xxx.cc" file (ex. "{path}/add.cc"), use g++/gcc command to compile it.(ex. "g++ –shared -fPIC -o add.so add.cc")

       * Define a "xxx.cc"/"xxx.cu" file:

         "aot" is a cross-platform identity. The functions defined in "xxx.cc" or "xxx.cu" share the same args. Typically, the function should be as:

         ```
         int func(int nparam, void **params, int *ndims, int64_t **shapes, const␣
         ↪char **dtypes,
                 void *stream, void *extra)
         ```

         Parameters:

         · nparam(int): total number of inputs plus outputs; suppose the operator has 2 inputs and 3 outputs, then nparam=5

         · params(void **): a pointer to the array of inputs and outputs' pointer; the pointer type of inputs and outputs is void * ; suppose the operator has 2 inputs and 3 outputs, then the first input's pointer is params[0] and the second output's pointer is params[3]

         · ndims(int *): a pointer to the array of inputs and outputs' dimension num; suppose params[i] is a 1024x1024 tensor and params[j] is a 77x83x4 tensor, then ndims[i]=2, ndims[j]=3.

         · shapes(int64_t **): a pointer to the array of inputs and outputs' shapes(int64_t *); the ith input's jth dimension's size is shapes[i][j](0<=j<ndims[i]); suppose params[i] is a 2x3 tensor and params[j] is a 3x3x4 tensor, then shapes[i][0]=2, shapes[j][2]=4.

         · dtypes(const char **): a pointer to the array of inputs and outputs' types(const char *); (ex. "float32", "float16", "float", "float64", "int", "int8", "int16", "int32", "int64", "uint", "uint8", "uint16", "uint32", "uint64", "bool")

         · stream(void *): stream pointer, only used in cuda file

         · extra(void *): used for further extension

         Return Value(int):

- 0: MindSpore will continue to run if this aot kernel is successfully executed
- others: MindSpore will raise exception and exit

Examples: see details in tests/st/ops/graph_kernel/custom/aot_test_files/

* Use it in Custom:

```
Custom(func="{dir_path}/{file_name}:{func_name}",...)
(ex. Custom(func="./reorganize.so:CustomReorganize", out_shape=[1], out_
↪dtype=mstype.float32,
"aot"))
```

b) Ascend platform. Before using Custom operators on the Ascend platform, users must first develop custom operators based on Ascend C and compile them. The complete development and usage process can refer to the tutorial AOT-Type Custom Operators(Ascend). By passing the name of the operator through the input parameter *func*, there are two usage methods based on the implementation of the infer function:

* Python infer: If the operator's infer function is implemented in Python, that is, the infer shape function is passed through the *out_shape* parameter, and the infer type is passed throuht the *out_dtype*, then the *func* should be specified as the operator name, for example, *func="CustomName"*.
* C++ infer: If the operator's infer function is implemented through C++, then pass the path of the infer function implementation file in *func* and separate the operator name with *:*, for example: *func="add_custom_infer.cc:AddCustom"* .

2. for "julia":

Currently "julia" supports CPU(linux only) platform. For julia use JIT compiler, and julia support c api to call julia code. The Custom can directly launches user defined "xxx.jl" file as an operator. Users need to write a "xxx.jl" file which include modules and functions, and offer the path of the file along with a module name and function name.

Examples: see details in tests/st/ops/graph_kernel/custom/julia_test_files/

* Use it in Custom:

```
Custom(func="{dir_path}/{file_name}:{module_name}:{func_name}",...)
(ex. Custom(func="./add.jl:Add:add", out_shape=[1], out_dtype=mstype.
↪float32, "julia"))
```

- **out_shape** (*Union[function, list, tuple]*) −The output shape infer function or the value of output shape of *func*. Default: `None` .

  If func has single output, then the value of output shape is a list or tuple of int.

  If func has multiple outputs, then the value of output shape is a tuple, each item represents the shape of each output.

  The input can be None only when the func_type input is "hybrid". In this case, the automatic infer shape mechanic will be enabled.

- **out_dtype** (Union[function, *mindspore.dtype*, tuple[*mindspore.dtype*]]) −The output data type infer function or the value of output data type of *func*. Default: `None` .

  If func has single output, then the value of output shape is a *mindspore.dtype*.

  If func has multiple outputs, then the value of output shape is a tuple of *mindspore.dtype*, each item represents the data type of each output.

  The input can be None only when the func_type input is "hybrid". In this case, the automatic infer value mechanic will be enabled.

- **func_type** (*str*) −The implementation type of *func*, should be one of

  ["aot" , "pyfunc" , "julia"].

- **bprop** (*function*) −The back propagation function of *func*. Default: None .

- **reg_info** (*Union[str, dict, list, tuple]*) −Represents the registration information(reg info) of *func* with json format of type str or dict. The reg info specifies supported data types and formats of inputs and outputs, attributes and target of *func*. Default: None .

  If reg info is a list or tuple, then each item should be with json format of type str or dict, which represents the registration information of *func* in a specific target. You need to invoke *CustomRegOp* or the subclass of *RegOp* to generate the reg info for *func*. Then you can invoke *custom_info_register* to bind the reg info to *func* or just pass the reg info to *reg_info* parameter. The *reg_info* parameter takes higher priority than *custom_info_register* and the reg info in a specific target will be registered only once.

  If reg info is not set, then we will infer the data types and formats from the inputs of *Custom* operator.

  Please note that, if *func_type* is "tbe" or the *func* only supports some specified data types and formats, or it has attribute inputs, then you should set the reg info for *func*.

Inputs:

- **input** (Union(tuple, list)) - The input tuple or list is made up of multiple tensors, and attributes value(optional).

Outputs:

Tensor or tuple[Tensor], execution results.

Raises

- **TypeError** −If the type of *func* is invalid or the type of register information for *func* is invalid.
- **ValueError** −If *func_type* is invalid.
- **ValueError** −If the register information is invalid, including the target is not supported, the input numbers or the attributes of *func* differs in different targets.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> from mindspore.ops import CustomRegOp, custom_info_register, DataType, kernel
>>> from mindspore import dtype as mstype
>>> from mindspore.nn import Cell
>>> input_x = Tensor(np.ones([16, 16]).astype(np.float32))
>>> input_y = Tensor(np.ones([16, 16]).astype(np.float32))
>>>
>>> # Example, func_type = "pyfunc"
>>> def func_pyfunc(x1, x2):
...     return x1 + x2
>>>
>>> test_pyfunc = ops.Custom(func_pyfunc, lambda x, _: x, lambda x, _: x, "pyfunc")
>>> output = test_pyfunc(input_x, input_y)
>>> print(output.shape)
(16, 16)
```

18.12.2 mindspore.ops.CustomOpBuilder

class mindspore.ops.**CustomOpBuilder**(*name*, *sources*, *backend=None*, *include_paths=None*, *cflags=None*, *ldflags=None*, ***kwargs*)

CustomOpBuilder is used to initialize and configure custom operators for MindSpore. Users can define and load custom operator modules through this class and apply them to the network.

In most cases, users only need to provide the source files and additional compilation options in the constructor and call the *load* method to complete the compilation and loading of the operator. If users have specific customization requirements, they can inherit this class and override certain methods. It is important to note that if methods are overridden, some parameters passed to the constructor may be ignored.

Warning: This is an experimental API that is subject to change.

Parameters

- **name** (*str*) –The unique name of the custom operator module, used to identify the operator.
- **sources** (*Union[str, list[str]]*) –The source file(s) of the custom operator. It can be a single file path or a list of file paths.
- **backend** (*str, optional*) –The target backend for the operator, such as "CPU" or "Ascend". Default: `None`.
- **include_paths** (*list[str], optional*) –Additionally included paths needed during compilation. Default: `None`.
- **cflags** (*str, optional*) –Extra C++ compiler flags to be used during compilation. Default: `None`.
- **ldflags** (*str, optional*) –Extra linker flags to be used during linking. Default: `None`.
- **kwargs** (*dict, optional*) –Additional keyword arguments for future extensions or specific custom requirements.

Note:

- If the *backend* argument is provided, additional default flags will be automatically added to the compilation and linking steps to support the operator's target backend. The default options can be referenced in the implementation of the *get_cflags* and *get_ldflags* methods in the CustomOpBuilder.
- The *sources* argument must point to valid source files for the custom operator.

Supported Platforms:

`Ascend` `CPU`

Examples

```
>>> from mindspore import ops
>>> builder = ops.CustomOpBuilder(
...     name="custom_op_cpu",
...     sources="custom_ops_impl/pybind_op_cpu.cpp",
...     backend="CPU"
... )
>>> my_ops = builder.load()
```

build()
Build the custom operator module.

This method generates a dynamic library file for the custom operator based on the provided source files, include paths, compilation flags, and link flags.

Returns

str, The path to the compiled module.

get_cflags()
Get the C++ compiler flags for building the custom operator.

Returns

list[str], A list of C++ compiler flags.

get_include_paths()
Get the include paths required for compiling the custom operator.

Returns

list[str], A list of include paths.

get_ldflags()
Get the linker flags for building the custom operator.

Returns

list[str], A list of linker flags.

get_sources()
Get the source files for the custom operator.

Returns

str or list[str], The source file(s) for the operator.

load()
Build and load the custom operator module.

Returns

Module, The loaded custom operator module.

18.12.3 mindspore.ops.custom_info_register

mindspore.ops.**custom_info_register**(**reg_info*)
A decorator which is used to bind the registration information to the *func* parameter of *mindspore.ops.Custom*.

Note: The 'reg_info' will be added into oplib.

Parameters

reg_info (*tuple[str, dict]*) – Each item represents registration information in json format.

Returns

Function, returns a decorator for op info register.

Raises

TypeError – If *reg_info* is not a tuple.

Supported Platforms:

`Ascend` `GPU` `CPU`

Examples

```
>>> from mindspore.ops import custom_info_register, CustomRegOp, DataType
>>> custom_func_ascend_info = CustomRegOp() \
...     .input(0, "x", "dynamic") \
...     .output(0, "y") \
...     .dtype_format(DataType.F16_Default, DataType.F16_Default) \
...     .dtype_format(DataType.F32_Default, DataType.F32_Default) \
...     .target("Ascend") \
...     .get_op_info()
>>>
>>> @custom_info_register(custom_func_ascend_info)
... def custom_func(x):
...     pass
```

18.12.4 mindspore.ops.kernel

`mindspore.ops.`**`kernel`**(*fn=None*, *reg_info=None*, *compile_attrs=None*)

The decorator of the Hybrid DSL function for the Custom Op. When a function written by the Hybrid DSL is decorated by kernel, it can be run as a usual Python function. Also, this function can be used in the api Custom and to create *mindspore.ops.Custom*, with func_type "hybrid" or "pyfunc". Creating *mindspore.ops.Custom* with mode "hybrid" by the Hybrid DSL function will enjoy the automatic dtype/shape infer for free.

Parameters

- **fn** (*Function*) –The Python function that will be run as a custom operator. Default: `None` .
- **reg_info** (*tuple[str, dict]*) –Each item represents registration information in json format. Default: `None` .
- **compile_attrs** (*Dict*) –The Python object is used to distinguish the compiled function. Default: `None` .

Returns

Function, if *fn* is not None, returns a callable function that will execute the Hybrid DSL function; If *fn* is None, returns a decorator and when this decorator invokes with a single *fn* argument, the callable function is equal to the case when *fn* is not None.

Supported Platforms:

`GPU` `CPU`

Examples

```
>>> import numpy as np
>>> from mindspore import ops, Tensor
>>> from mindspore.ops import kernel, DataType, CustomRegOp
...
>>> # Create a dict for the compile flags.
>>> attrs = {
...     "test1": True,
...     "test2": "good",
```

(continues on next page)

(continued from previous page)

```
...     "test3": 12,
... }
>>> # Create the reg info json string.
>>> op_cpu_info = CustomRegOp() \
...     .input(0, "a") \
...     .input(0, "b") \
...     .output(0, "y") \
...     .dtype_format(DataType.F32_None, DataType.F32_None, DataType.F32_None) \
...     .target("CPU") \
...     .get_op_info()
>>>
>>> # Create inputs for the custom op.
>>> input_x = np.ones([4, 4]).astype(np.float32)
>>> input_y = np.ones([4, 4]).astype(np.float32)
...
>>> # Write a Hybrid DSL function through the decorator @kernel.
>>> # We can also pass the compile attrs and the reg info through the decorator.
>>> @kernel(reg_info=op_cpu_info, compile_attrs=attrs)
... def outer_product(a, b):
...     c = output_tensor(a.shape, a.dtype)
...
...     with block_realize(c):
...         for i0 in range(a.shape[0]):
...             for i1 in range(b.shape[1]):
...                 c[i0, i1] = 0.0
...                 for i2 in range(a.shape[1]):
...                     c[i0, i1] = c[i0, i1] + (a[i0, i2] * b[i2, i1])
...     return c
...
>>> # We can use the function directly as a python function.
>>> # In this case, the inputs should be numpy arrays.
>>> result = outer_product(input_x, input_y)
```

18.13 Spectral Operator

API Name	Description	Supported Platforms
mindspore.ops.BartlettWindow	Bartlett window function.	Ascend GPU CPU
mindspore.ops.BlackmanWindow	Blackman window function.	Ascend GPU CPU

18.13.1 mindspore.ops.BartlettWindow

class mindspore.ops.**BartlettWindow**(*periodic=True*, *dtype=mstype.float32*)

Bartlett window function.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.bartlett_window()* for more details.

Parameters

- **periodic** (*bool, optional*) –If True, returns a window to be used as periodic function. If False, return a symmetric window. Default: `True`.
- **dtype** (*mindspore.dtype, optional*) –The desired datatype of returned tensor. Only float16, float32 and float64 are allowed. Default: mstype.float32.

Inputs:

- **window_length** (Tensor) - The size of returned window, with data type int32, int64. The input data should be an integer with a value of [0, 1000000].

Outputs:

A 1-D tensor of size *window_length* containing the window. Its datatype is set by the attr *dtype*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> from mindspore import Tensor, ops
>>> from mindspore import dtype as mstype
>>> window_length = Tensor(5, mstype.int32)
>>> bartlett_window = ops.BartlettWindow(periodic=True, dtype=mstype.float32)
>>> output = bartlett_window(window_length)
>>> print(output)
[0.  0.4 0.8 0.8 0.4]
```

18.13.2 mindspore.ops.BlackmanWindow

class mindspore.ops.**BlackmanWindow**(*periodic=True*, *dtype=mstype.float32*)

Blackman window function.

> **Warning:** This is an experimental API that is subject to change or deletion.

Refer to *mindspore.ops.blackman_window()* for more details.

Parameters

- **periodic** (*bool, optional*) –If `True` , returns a window to be used as periodic function. If `False` , return a symmetric window. Default: `True` .
- **dtype** (*mindspore.dtype, optional*) –the desired data type of returned tensor. Only float16, float32 and float64 is allowed. Default: `mstype.float32` .

Inputs:

- **window_length** (Tensor) - the size of returned window, with data type int32, int64. The input data should be an integer with a value of [0, 1000000].

Outputs:

A 1-D tensor of size *window_length* containing the window. Its datatype is set by the attr *dtype*.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> from mindspore import Tensor, ops
>>> window_length = Tensor(10, mindspore.int32)
>>> blackman_window = ops.BlackmanWindow(periodic = True, dtype = mindspore.float32)
>>> output = blackman_window(window_length)
>>> print(output)
[-2.9802322e-08  4.0212840e-02  2.0077014e-01  5.0978714e-01
  8.4922993e-01  1.0000000e+00  8.4922981e-01  5.0978690e-01
  2.0077008e-01  4.0212870e-02]
```

CHAPTER

NINETEEN

MINDSPORE.BOOST

Boost is able to automatically optimize network performance, e.g., by reducing BN, gradient freezing, and accumulating gradients to achieve network acceleration.

Note: This feature is a beta feature, and we are still improving its functionality.

class mindspore.boost.**AdaSum**(*rank*, *device_number*, *group_number*, *parameter_tuple*)

The Adaptive Summation, or AdaSum, is a novel algorithm for improving distributed data parallel training of Deep Learning models.

Parameters

- **rank** (*int*) –Rank number.
- **device_number** (*int*) –Device number.
- **group_number** (*int*) –Group number.
- **parameter_tuple** (*Tuple(Parameter)*) –Tuple of parameters.

Inputs:

- **delta_weights** (Tuple(Tensor)) - Tuple of gradients.
- **parameters** (Tuple(Parameter)) - Tuple of current parameters.
- **old_parameters** (Tuple(Parameter)) - Tuple of last parameters.

Outputs:

- **adasum_parameters** (Tuple(Tensor)) - Tuple of parameters after adasum process.

class mindspore.boost.**AutoBoost**(*level='O0'*, *boost_config_dict=''*)

Provide auto accelerating for network.

Parameters

- **level** (*str*) –Boost config level. Default: "O0" .
- **boost_config_dict** (*dict*) –User config hyperparameter dict, recommended config format:

```
{
    "boost": {
        "mode": "auto",
        "less_bn": False,
        "grad_freeze": False,
        "adasum": False,
```

(continues on next page)

(continued from previous page)

```
        "grad_accumulation": False,
        "dim_reduce": False,
        "loss_scale_group": False
    },
    "common": {
        "gradient_split_groups": [50, 100],
        "device_number": 8
    },
    "less_bn": {
        "fn_flag": True,
        "gc_flag": True
    },
    "grad_freeze": {
        "param_groups": 10,
        "freeze_type": 1,
        "freeze_p": 0.7,
        "total_steps": 65536
    }
    "dim_reduce": {
        "rho": 0.55,
        "gamma": 0.9,
        "alpha": 0.001,
        "sigma": 0.4,
        "n_components": 32,
        "pca_mat_path": None,
        "weight_load_dir": None,
        "timeout": 1800
    }
}
```

Default: `""` .

- boost:
  - mode (str): How to set the boost. Supports ["auto", "manual", "enable_all", "disable_all"]. Default: `"auto"` .
    - auto: Depend on the argument "boost_level" in class Model.
    - manual: Depend on "boost_config_dict".
    - enable_all: Set all boost functions true.
    - disable_all: Set all boost functions false.
  - less_bn (bool): Whether to apply less_bn function. Default: `False` .
  - grad_freeze: (bool): Whether to apply grad_freeze function. Default: `False` .
  - adasum (bool): Whether to apply adasum function. Default: `False` .
  - grad_accumulation (bool): Whether to apply grad_accumulation function. Default: `False` .
  - dim_reduce (bool): Whether to apply dim_reduce function. Default: `False` .
  - loss_scale_group (bool): Whether to apply loss_scale_group function. Default: `False` .

  If set dim_reduce true, other functions will be false. If set grad_freeze true and dim_reduce false, other functions will be false.
- common:
  - gradient_split_groups (list): The gradient split point of this network. Default: `[50, 100]` .

    - device_number (int): Device number. Default: `8`.

  - less_bn:
    - fn_flag (bool): Whether changing fc to fn. Default: `True`.
    - gc_flag (bool): Whether to apply gc. Default: `True`.
  - grad_freeze:
    - param_groups (int): The number of parameter groups. Default: `10`.
    - freeze_type (int): Gradient freeze grouping strategy, select from [0, 1]. Default: `1`.
    - freeze_p (float): Gradient freezing probability. Default: `0.7`.
    - total_steps (int): Total training steps. Default: `65536`.
  - dim_reduce:

    The leading principles of dim_reduce:

$$\begin{aligned} grad\_k &= pca\_mat \cdot grad \\ dk &= -bk \cdot grad\_k \\ sk &= rho^m \cdot dk \\ delta\_loss &= sigma \cdot grad\_k.T \cdot sk \end{aligned}$$

    Here:
    - pca_mat (array): Shape $(k * n)$, k is part of n_components, n is the size of weight.
    - bk (array): Shape $(k * k)$, is the symmetric positive definite matrix in Quasi-Newton method.

    we need to find the m satisfy:

$$new\_loss < old\_loss + delta\_loss$$

    Then, get delta_grad to update the weights for model:

$$\begin{aligned} grad\_k\_proj &= pca\_mat.T \cdot grad\_k \\ new\_grad\_momentum &= gamma \cdot old\_grad\_momentum + grad - grad\_k\_proj \\ delta\_grad &= alpha \cdot new\_grad\_momentum - pca\_mat.T \cdot sk \end{aligned}$$

    - rho (float): Generally, it does not need to be modified. Default: `0.55`.
    - gamma (float): Generally, it does not need to be modified. Default: `0.9`.
    - alpha (float): Generally, it does not need to be modified. Default: `0.001`.
    - sigma (float): Generally, it does not need to be modified. Default: `0.4`.
    - n_components (int): PCA component. Default: `32`.
    - pca_mat_path (str): The path to load pca mat. Default: `None`.
    - weight_load_dir (str): The directory to load weight files saved as ckpt. Default: `None`.
    - timeout (int): Waiting time to load local pca mat. Default: `1800 (second)`.

  User can load the config through the JSON file or use the dictionary directly. The unconfigured parameters will adopt the default values.

Raises

ValueError – The boost mode not in ["auto", "manual", "enable_all", "disable_all"].

Supported Platforms:

`Ascend`

Examples

```
>>> from mindspore.boost import AutoBoost
>>> #1) when configuring the dict directly:
>>> boost_config_dict = {"boost": {"mode": "auto"}}
>>> boost = AutoBoost("O1", boost_config_dict)
>>>
>>> #2) when loading the dict from a json file:
>>> import json
>>> boost_json = "/path/boost_config.json"
>>> with open(boost_json, 'r') as fp:
...     boost_config_dict = json.load(fp)
>>> boost = AutoBoost("O1", boost_config_dict)
```

network_auto_process_eval(*network*)
Boost network eval.

Parameters

network (Cell) –The inference network.

network_auto_process_train(*network*, *optimizer*)
Boost network train.

Parameters

- **network** (Cell) –The training network.
- **optimizer** (Cell) –Optimizer for updating the weights.

class mindspore.boost.**BoostTrainOneStepCell**(*network*, *optimizer*, *sens=None*)
Boost Network training package class.

Wraps the network with an optimizer. The resulting Cell is trained with input '*inputs'. The backward graph will be created in the construct function to update the parameter, and different parallel modes are available for training.

Parameters

- **network** (Cell) –The training network. The network only supports single output.
- **optimizer** (*Union[Cell]*) –Optimizer for updating the weights.
- **sens** (*numbers.Number*) –The scaling number to be filled as the input of backpropagation. Default: None , which is 1.0 .

Inputs:

- ***inputs** (Tuple(Tensor)) - Tuple of input tensors with shape (N, ...).

Outputs:

Tensor, a tensor means the loss value, the shape of which is usually ().

- loss(Tensor): A scalar Tensor.
- overflow(Tensor): A scalar Tensor which type is bool.
- loss scaling value(Tensor): A scalar Tensor.

Raises

TypeError –If *sens* is not a number.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> from mindspore import boost
>>> from mindspore import nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> loss_fn = nn.SoftmaxCrossEntropyWithLogits()
>>> optim = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> #1) Using the WithLossCell existing provide
>>> loss_net = nn.WithLossCell(net, loss_fn)
>>> train_net = boost.BoostTrainOneStepCell(loss_net, optim)
>>>
>>> #2) Using user-defined WithLossCell
>>> class MyWithLossCell(nn.Cell):
...     def __init__(self, backbone, loss_fn):
...         super(MyWithLossCell, self).__init__(auto_prefix=False)
...         self._backbone = backbone
...         self._loss_fn = loss_fn
...
...     def construct(self, x, y, label):
...         out = self._backbone(x, y)
...         return self._loss_fn(out, label)
...
...     @property
...     def backbone_network(self):
...         return self._backbone
...
>>> loss_net = MyWithLossCell(net, loss_fn)
>>> train_net = boost.BoostTrainOneStepCell(loss_net, optim)
```

adasum_process(*loss*, *grads*)
Adasum algorithm process.

Parameters

- **loss** (Tensor) –Tensor with shape ().
- **grads** (*tuple(Tensor)*) –Tuple of gradient tensors.

Returns

- **loss** (Tensor) - Network loss, tensor with shape ().

check_adasum_enable()
Check adasum enable.

Returns

- **enable_adasum** (bool) - Check whether the Adasum algorithm is enabled.

check_dim_reduce_enable()
Check dim_reduce enable.

Returns

- **enable_dim_reduce** (bool) - Check whether the dimensionality reduction second-order training algorithm is enabled.

gradient_accumulation_process (*loss*, *grads*, *sens*, **inputs*)
Gradient accumulation algorithm process.

Parameters

- **loss** (Tensor) −Tensor with shape ().
- **grads** (*tuple(Tensor)*) −Tuple of gradient tensors.
- **sens** (Tensor) −Tensor with shape ().
- **inputs** (*tuple(Tensor)*) −Tuple of input tensors with shape $(N, \ldots)$.

Returns

- **loss** (Tensor) - Network loss, tensor with shape ().

gradient_freeze_process (**inputs*)
Gradient freeze algorithm process.

Parameters

inputs (*tuple(Tensor)*) −Tuple of input tensors with shape $(N, \ldots)$.

Returns

- **loss** (Tensor) - Network loss, tensor with shape ().

class mindspore.boost.**BoostTrainOneStepWithLossScaleCell** (*network*, *optimizer*, *scale_sense*)
Boost Network training with loss scaling.

This is a training step with loss scaling. It takes a network, an optimizer and possibly a scale update Cell as args. The loss scale value can be updated in both host side or device side. The BoostTrainOneStepWithLossScaleCell will be compiled to be graph which takes **inputs* as input data. The Tensor type of *scale_sense* is acting as loss scaling value. If you want to update it on host side, the value must be provided. If the Tensor type of *scale_sense* is not given, the loss scale update logic must be provide by Cell type of *scale_sense*.

Parameters

- **network** (Cell) −The training network. The network only supports single output.
- **optimizer** (Cell) −Optimizer for updating the weights.
- **scale_sense** (*Union[Tensor, Cell]*) −If this value is Cell type, the loss scaling update logic cell.If this value is Tensor type, *mindspore.nn.TrainOneStepWithLossScaleCell.set_sense_scale()* can be called to update loss scale factor, Tensor with shape () or $(1,)$.

Inputs:

- ***inputs** (Tuple(Tensor)) - Tuple of input tensors with shape $(N, \ldots)$.

Outputs:

Tuple of 3 Tensor, the loss, overflow flag and current loss scaling value.

- **loss** (Tensor) - Tensor with shape ().
- **overflow** (Tensor) - Tensor with shape (), type is bool.
- **loss scaling value** (Tensor) - Tensor with shape ()

Raises

- **TypeError** −If *scale_sense* is neither Cell nor Tensor.
- **ValueError** −If shape of *scale_sense* is neither $(1,)$ nor ().

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.nn import WithLossCell
>>> from mindspore import dtype as mstype
>>> from mindspore import boost
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
 ↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
...
>>> size, in_features, out_features = 16, 16, 10
>>> #1) when the type of scale_sense is Cell:
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = WithLossCell(net, loss)
>>> manager = nn.DynamicLossScaleUpdateCell(loss_scale_value=2**12, scale_factor=2, scale_
 ↪window=1000)
>>> train_network = boost.BoostTrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
 ↪sense=manager)
>>> input = Tensor(np.ones([out_features, in_features]), mstype.float32)
>>> labels = Tensor(np.ones([out_features,]), mstype.float32)
>>> output = train_network(input, labels)
>>>
>>> #2) when the type of scale_sense is Tensor:
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = WithLossCell(net, loss)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.zeros([size, out_features]).astype(np.float32))
>>> scaling_sens = Tensor(np.full((1), np.finfo(np.float32).max), dtype=mstype.float32)
>>> train_network = boost.BoostTrainOneStepWithLossScaleCell(net_with_loss, optimizer, scale_
 ↪sense=scaling_sens)
>>> output = train_network(inputs, label)
```

class mindspore.boost.**DimReduce**(*network*, *optimizer*, *weight*, *pca_mat_local*, *n_components*, *rho*, *gamma*, *alpha*, *sigma*, *rank*, *rank_size*)

The dimension reduce training, is a novel algorithm for accelerating convergence of Deep Learning models.

$$\begin{aligned} grad\_k &= pca\_mat \cdot grad \\ dk &= -bk \cdot grad\_k \\ sk &= rho^m \cdot dk \\ delta\_loss &= sigma \cdot grad\_k.T \cdot sk \end{aligned}$$

Here:

- pca_mat (array): Shape $(k * n)$, k is part of n_components, n is the size of weight.
- bk (array): Shape $(k * k)$, is the symmetric positive definite matrix in Quasi-Newton method.

we need to find the m satisfy:

$$new\_loss < old\_loss + delta\_loss$$

Then, get delta_grad to update the weights for model:

$$\begin{aligned} grad\_k\_proj &= pca\_mat.T \cdot grad\_k \\ new\_grad\_momentum &= gamma \cdot old\_grad\_momentum + grad - grad\_k\_proj \\ delta\_grad &= alpha \cdot new\_grad\_momentum - pca\_mat.T \cdot sk \end{aligned}$$

Parameters

- **network** (Cell) −The training network. The network only supports single output.
- **optimizer** (*Union[Cell]*) −Optimizer for updating the weights.
- **weight** (*Tuple(Parameter)*) −Tuple of parameters.
- **pca_mat_local** (*numpy.ndarray*) −For PCA operation, k*n, k is part of n_components, n is the size of weight.
- **n_components** (*int*) −PCA.components.
- **rho** (*float*) −Coefficient.
- **gamma** (*float*) −Coefficient.
- **alpha** (*float*) −Coefficient.
- **sigma** (*float*) −Coefficient.
- **rank** (*int*) −Rank number.
- **rank_size** (*int*) −Rank size.

Inputs:

- **loss** (Tensor) - Tensor with shape ().
- **old_grad** (Tuple(Tensor)) - Tuple of gradient tensors.
- **weight** (Tuple(Tensor)) - Tuple of parameters.
- **weight_clone** (Tuple(Tensor)) - clone of weight
- ***inputs** (Tuple(Tensor)) - Tuple of input tensors with shape $(N, \ldots)$.

Outputs:

- **loss** (Tensor) - Tensor with shape ().

class mindspore.boost.**FreezeOpt**(*opt*, *train_parameter_groups=None*, *train_strategy=None*)

Optimizer that supports gradients freezing training.

Parameters

- **opt** (Cell) –non-freezing optimizer instance, such as 'Momentum', 'SGD'.
- **train_parameter_groups** (*Union[tuple, list]*) –Groups of parameters for gradients freezing training. Default: None .
- **train_strategy** (*Union[tuple(int), list(int), Tensor]*) –Strategy for gradients freezing training. Default: None .

Supported Platforms:

Ascend

class mindspore.boost.**GradientAccumulation**(*max_accumulation_step*, *optimizer*)

After accumulating the gradients of multiple steps, call to optimize its update.

Parameters

- **max_accumulation_step** (*int*) –Steps to accumulate gradients.
- **optimizer** (Cell) –Optimizer used.

class mindspore.boost.**GradientFreeze**(*param_groups*, *freeze_type*, *freeze_p*, *total_steps*)

Gradients freezing algorithm, freezing the gradients of some layers randomly, to improve network training performance. The number and probability of frozen layers can be configured by users.

Parameters

- **param_groups** (*Union[tuple, list]*) –Groups of parameters for gradients freezing training.
- **freeze_type** (*int*) –Strategy of gradients freezing training.
- **freeze_p** (*float*) –probability of gradients freezing training.
- **total_steps** (*int*) –Steps of the whole training.

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.nn import WithLossCell
>>> from mindspore import dtype as mstype
>>> from mindspore import boost
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
>>> size, in_features, out_features = 16, 16, 10
```

(continues on next page)

(continued from previous page)

```
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = WithLossCell(net, loss)
>>> gradient_freeze_class = boost.GradientFreeze(10, 1, 0.5, 2000)
>>> network, optimizer = gradient_freeze_class.freeze_generate(net_with_loss, optimizer)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.zeros([size, out_features]).astype(np.float32))
>>> output = network(inputs, label)
```

freeze_generate(*network*, *optimizer*)
Generate freeze network and optimizer.

Parameters

- **network** (Cell) –The training network.
- **optimizer** (Cell) –Optimizer for updating the weights.

generate_freeze_index_sequence(*parameter_groups_number*, *freeze_strategy*, *freeze_p*, *total_steps*)
Generate index sequence for gradient freezing training.

Parameters

- **parameter_groups_number** (*int*) –The number of parameter groups.
- **freeze_strategy** (*int*) –Gradient freeze grouping strategy, select from [0, 1].
- **freeze_p** (*float*) –Gradient freezing probability.
- **total_steps** (*int*) –Total training steps.

split_parameters_groups(*net*, *freeze_para_groups_number*)
Split parameter groups for gradients freezing training.

Parameters

- **net** (Cell) –The training network.
- **freeze_para_groups_number** (*int*) –The number of gradient freeze groups.

class mindspore.boost.**GroupLossScaleManager**(*init_loss_scale*, *loss_scale_groups*)
Enhanced hybrid precision algorithm supports multi-layer application of different loss scales and dynamic updating of loss scales.

Parameters

- **init_loss_scale** (*Number*) –The initialized loss scale value.
- **loss_scale_groups** (*List*) –The loss scale groups, which are divided from the param list.

Inputs:

- **x** (Tensor) - The output of last operator.
- **layer1** (Int) - Current network layer value.
- **layer2** (Int) - Last network layer value.

Outputs:

- **out** (Tensor) - A tensor with a group of loss scale tags that marks the loss scale group number of the current tensor.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore as ms
>>> from mindspore import boost, nn
>>>
>>> class Net(nn.Cell):
...     def __init__(self, enhanced_amp, num_class=10, num_channel=1):
...         super(Net, self).__init__()
...         self.conv1 = nn.Conv2d(num_channel, 6, 5, pad_mode='valid')
...         self.conv2 = nn.Conv2d(6, 16, 5, pad_mode='valid')
...         self.fc1 = nn.Dense(16*5*5, 120, weight_init='ones')
...         self.fc2 = nn.Dense(120, 84, weight_init='ones')
...         self.fc3 = nn.Dense(84, num_class, weight_init='ones')
...         self.relu = nn.ReLU()
...         self.max_pool2d = nn.MaxPool2d(kernel_size=2, stride=2)
...         self.flatten = nn.Flatten()
...         self.enhanced_amp = enhanced_amp
...
...     def construct(self, x):
...         x = self.enhanced_amp(x, 0, 1)
...         x = self.max_pool2d(self.relu(self.conv1(x)))
...         x = self.max_pool2d(self.relu(self.conv2(x)))
...         x = self.flatten(x)
...         x = self.enhanced_amp(x, 1, 2)
...         x = self.relu(self.fc1(x))
...         x = self.relu(self.fc2(x))
...         x = self.fc3(x)
...         x = self.enhanced_amp(x, 2, 3)
...         return x
>>>
>>> loss_scale_manager = boost.GroupLossScaleManager(4096, [])
>>> net = Net(loss_scale_manager)
>>> param_group1 = []
>>> param_group2 = []
>>> for param in net.trainable_params():
...     if 'conv' in param.name:
...         param_group1.append(param)
...     else:
...         param_group2.append(param)
>>> loss_scale_manager.loss_scale_groups = [param_group1, param_group2]
>>> loss = nn.SoftmaxCrossEntropyWithLogits()
>>> optim = nn.Momentum(params=net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> boost_config_dict = {"boost": {"mode": "manual", "less_bn": False, "grad_freeze": False,
↪"adasum": False,
...                       "grad_accumulation": False, "dim_reduce": False, "loss_scale_group":
↪ True}}
>>> model = ms.train.Model(net, loss_fn=loss, optimizer=optim, metrics=None,
...                        loss_scale_manager=loss_scale_manager,
...                        boost_level="O1", boost_config_dict=boost_config_dict)
>>> # Create the dataset taking MNIST as an example. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/mnist.py
>>> dataset = create_dataset()
>>> model.train(2, dataset)
```

get_loss_scale()

Get loss scale value.

Returns

bool, *loss_scale* value.

get_update_cell()
Returns the instance of *mindspore.boost.GroupLossScaleManager*.

Returns

mindspore.boost.GroupLossScaleManager.

set_loss_scale_status(*loss_scale_number*, *init_loss_scale*)
Generate dynamic loss scale tuple and set overflow status list.

Parameters

- **loss_scale_number** (*int*) –The number of loss scale.
- **init_loss_scale** (*float*) –The initialized loss scale.

update_loss_scale_status(*layer*, *update_ratio*)
Update dynamic loss scale.

Parameters

- **layer** (*int*) –Current layer.
- **update_ratio** (*float*) –The ratio of loss scale update.

Outputs:

float, new loss scale.

class mindspore.boost.**LessBN**(*network*, *fn_flag=False*)
Reduce the number of BN automatically to improve the network performance and ensure the network accuracy.

Parameters

- **network** (Cell) –Network to be modified.
- **fn_flag** (*bool*) –Replace FC with FN. Default: False .

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.nn import WithLossCell
>>> from mindspore import dtype as mstype
>>> from mindspore import boost
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
>>> size, in_features, out_features = 16, 16, 10
```

(continues on next page)

(continued from previous page)

```
>>> net = Net(in_features, out_features)
>>> loss = nn.MSELoss()
>>> optimizer = nn.Momentum(net.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> net_with_loss = WithLossCell(net, loss)
>>> inputs = Tensor(np.ones([size, in_features]).astype(np.float32))
>>> label = Tensor(np.zeros([size, out_features]).astype(np.float32))
>>> train_network = boost.LessBN(net_with_loss)
>>> output = train_network(inputs, label)
```

class mindspore.boost.**OptimizerProcess**(*opt*)

Process optimizer for Boost. Currently, this class supports adding GC(grad centralization) tags and creating new optimizers.

Parameters

opt (Cell) – Optimizer used.

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.boost import OptimizerProcess
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
...
>>> size, in_features, out_features = 16, 16, 10
>>> network = Net(in_features, out_features)
>>> optimizer = nn.Momentum(network.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> optimizer_process = OptimizerProcess(optimizer)
>>> optimizer_process.add_grad_centralization(network)
>>> optimizer = optimizer_process.generate_new_optimizer()
```

add_grad_centralization(*network*)

Add gradient centralization.

Parameters

network (Cell) – The training network.

static build_gc_params_group(*params_dict*, *parameters*)

Build the parameter's group with grad centralization.

Parameters

- **params_dict** (*dict*) – The network's parameter dict.
- **parameters** (*list*) – The network's parameter list.

static build_params_dict(*network*)
Build the parameter's dict of the network.

Parameters

network (Cell) –The training network.

generate_new_optimizer()
Generate new optimizer.

class mindspore.boost.**ParameterProcess**
Process parameter for Boost. Currently, this class supports creating group parameters and automatically setting gradient segmentation point.

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.boost import ParameterProcess
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.weight2 = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight2')
...         self.matmul = ops.MatMul()
...         self.matmul2 = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         output2 = self.matmul2(x, self.weight2)
...         return output + output2
...
>>> size, in_features, out_features = 16, 16, 10
>>> network = Net(in_features, out_features)
>>> new_parameter = network.trainable_params()[:1]
>>> group_params = ParameterProcess.generate_group_params(new_parameter, network.trainable_
↪params())
```

assign_parameter_group(*parameters*, *split_point=None*)
Assign parameter group.

Parameters

- **parameters** (*list*) –The network's parameter list.
- **split_point** (*list*) –The gradient split point of this network. Default: None.

static generate_group_params(*parameters*, *origin_params*)
Generate group parameters.

Parameters

- **parameters** (*list*) –The network's parameter list.
- **origin_params** (*list*) –The network's origin parameter list.

mindspore.boost.**freeze_cell**(*reducer_flag*, *network*, *optimizer*, *sens*, *grad*, *use_grad_accumulation*, *mean=None*, *degree=None*, *max_accumulation_step=1*)

Generate freeze network and optimizer.

Parameters

- **reducer_flag** (*bool*) –Reducer flag.
- **network** (Cell) –The training network.
- **optimizer** (Cell) –Optimizer for updating the weights.
- **sens** (*numbers.Number*) –The scaling number.
- **grad** (*tuple(Tensor)*) –Tuple of gradient tensors.
- **use_grad_accumulation** (*bool*) –Use gradient accumulation flag.
- **mean** (*bool*) –Gradients mean flag. Default: None .
- **degree** (*int*) –Device number. Default: None .
- **max_accumulation_step** (*int*) –Max accumulation steps. Default: 1 .

Examples

```
>>> import numpy as np
>>> from mindspore import Tensor, Parameter, nn
>>> from mindspore import ops
>>> from mindspore.boost.grad_freeze import freeze_cell, FreezeOpt
>>>
>>> class Net(nn.Cell):
...     def __init__(self, in_features, out_features):
...         super(Net, self).__init__()
...         self.weight = Parameter(Tensor(np.ones([in_features, out_features]).astype(np.
↪float32)),
...                                 name='weight')
...         self.matmul = ops.MatMul()
...
...     def construct(self, x):
...         output = self.matmul(x, self.weight)
...         return output
...
>>> in_features, out_features = 16, 10
>>> network = Net(in_features, out_features)
>>> optimizer = nn.Momentum(network.trainable_params(), learning_rate=0.1, momentum=0.9)
>>> optimizer = FreezeOpt(optimizer)
>>> grad = ops.GradOperation(get_by_list=True, sens_param=True)
>>> freeze_nets = freeze_cell(False, network, optimizer, 1.0, grad, False, None, None, 1)
```

CHAPTER

TWENTY

MINDSPORE.NN.PROBABILITY

20.1 Bijectors

API Name	Description	Supported Platforms
mindspore.nn.probability.bijector.Bijector	Bijecotr class.	Ascend GPU
mindspore.nn.probability.bijector.Exp	Exponential Bijector.	Ascend GPU
mindspore.nn.probability.bijector.GumbelCDF	GumbelCDF Bijector.	Ascend GPU
mindspore.nn.probability.bijector.Invert	Invert Bijector.	Ascend GPU CPU
mindspore.nn.probability.bijector.PowerTransform	PowerTransform Bijector.	Ascend GPU
mindspore.nn.probability.bijector.ScalarAffine	Scalar Affine Bijector.	Ascend GPU
mindspore.nn.probability.bijector.Softplus	Softplus Bijector.	Ascend GPU CPU

20.1.1 mindspore.nn.probability.bijector.Bijector

class mindspore.nn.probability.bijector.**Bijector**(*is_constant_jacobian=False*, *is_injective=True*, *name=None*, *dtype=None*, *param=None*)

Bijecotr class. A bijector perform a mapping from one distribution to the other via some function. If X is a random variable following the original distribution, and $g(x)$ is the mapping function, then $Y = g(X)$ is the random variable following the transformed distribution.

Parameters

- **is_constant_jacobian** (*bool*) −Whether the Bijector has constant derivative. Default: `False` .
- **is_injective** (*bool*) −Whether the Bijector is a one-to-one mapping. Default: `True` .
- **name** (*str*) −The name of the Bijector. Default: `None` .
- **dtype** (`mindspore.dtype`) −The type of the distributions that the Bijector can operate on. Default: `None` .
- **param** (*dict*) −The parameters used to initialize the Bijector. Default: `None` .

Note:

- *dtype* of bijector represents the type of the distributions that the bijector could operate on.

- When *dtype* is None, there is no enforcement on the type of input value except that the input value has to be float type. During initialization, when *dtype* is None, there is no enforcement on the dtype of the parameters. All parameters should have the same float type, otherwise a TypeError will be raised.

  Specifically, the parameter type will follow the dtype of the input value.

  – Parameters of the bijector will be casted into the same type as input value when *dtype* is None.

  – When *dtype* is specified, it is forcing the parameters and input value to be the same dtype as *dtype*. When the type of parameters or the type of the input value is not the same as *dtype*, a TypeError will be raised.

- Only subtype of mindspore.float_type can be used to specify bijector's *dtype*.

Supported Platforms:

`Ascend` `GPU`

cast_param_by_value(*value*, *para*)

Converts the data type of *para* in the input to the same type as *value*. Typically used by subclasses of Bijector to convert data types of their own parameters.

Parameters

- **value** (Tensor) −input value.
- **para** (Tensor) −parameter(s) of the bijector.

Returns

Tensor, the value of parameters after casting.

construct(*name*, **args*, ***kwargs*)

Override *construct* in Cell.

Note: Names of supported functions include: 'forward', 'inverse', 'forward_log_jacobian', and 'inverse_log_jacobian'.

Parameters

- **name** (*str*) −The name of the function.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Returns

Tensor, the result of the function corresponding to name.

forward(*value*, **args*, ***kwargs*)

Forward transformation: transform the input value to another distribution.

Parameters

- **value** (Tensor) −the value of the input variables.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Returns

Tensor, the value of the transformed random variable.

forward_log_jacobian(*value*, **args*, ***kwargs*)
Logarithm of the derivative of the forward transformation.

Parameters

- **value** (Tensor) −the value of the input variables.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Returns

Tensor, outputs the value of a random variable after mapping.

inverse(*value*, **args*, ***kwargs*)
Inverse transformation: transform the input value back to the original distribution.

Parameters

- **value** (Tensor) −the value of the transformed variables.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Returns

Tensor, the value of the input random variable.

inverse_log_jacobian(*value*, **args*, ***kwargs*)
Logarithm of the derivative of the inverse transformation.

Parameters

- **value** (Tensor) −the value of the transformed variables.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Returns

Tensor, the value of logarithm of the derivative of the inverse transformation.

20.1.2 mindspore.nn.probability.bijector.Exp

class mindspore.nn.probability.bijector.**Exp**(*name='Exp'*)
Exponential Bijector. This Bijector performs the operation:

$$Y = \exp(x).$$

Parameters

name (*str*) −The name of the Bijector. Default: 'Exp' .

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>>
>>> # To initialize an Exp bijector.
>>> exp_bijector = nn.probability.bijector.Exp()
>>> value = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> ans1 = exp_bijector.forward(value)
>>> print(ans1.shape)
(3,)
>>> ans2 = exp_bijector.inverse(value)
>>> print(ans2.shape)
(3,)
>>> ans3 = exp_bijector.forward_log_jacobian(value)
>>> print(ans3.shape)
(3,)
>>> ans4 = exp_bijector.inverse_log_jacobian(value)
>>> print(ans4.shape)
(3,)
```

forward(*value*)

forward mapping, compute the value after mapping as $Y = \exp(X)$.

Parameters

- **value** (Tensor) - the value to compute. X in the formula.

Returns

Tensor, the value of output after mapping.

forward_log_jacobian(*value*)

compute the log value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of forward mapping.

inverse(*value*)

Inverse mapping, compute the value after inverse mapping as $X = \log(Y)$.

Parameters

- **value** (Tensor) - the value of output after mapping. Y in the formula.

Returns

Tensor, the value to compute.

inverse_log_jacobian(*value*)

Compute the log value of the inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the log value of the inverse mapping.

20.1.3 mindspore.nn.probability.bijector.GumbelCDF

class mindspore.nn.probability.bijector.**GumbelCDF**(*loc=0.0*, *scale=1.0*, *name='GumbelCDF'*)

GumbelCDF Bijector. This Bijector performs the operation:

$$Y = \exp(-\exp(\frac{-(X - loc)}{scale}))$$

Parameters

- **loc** (*float*, *list*, *numpy.ndarray*, *Tensor*) – The location. Default: `0.0`.
- **scale** (*float*, *list*, *numpy.ndarray*, *Tensor*) – The scale. Default: `1.0`.
- **name** (*str*) – The name of the Bijector. Default: `'GumbelCDF'`.

Note:

- *scale* must be greater than zero.
- For *inverse* and *inverse_log_jacobian*, input should be in range of (0, 1).
- The dtype of *loc* and *scale* must be float.
- If *loc*, *scale* are passed in as numpy.ndarray or tensor, they have to have the same dtype otherwise an error will be raised.

Raises

TypeError – When the dtype of *loc* or *scale* is not float, or when the dtype of *loc* and *scale* is not same.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.bijector as msb
>>> from mindspore import Tensor
>>>
>>> # To initialize a GumbelCDF bijector of loc 1.0, and scale 2.0.
>>> gumbel_cdf = msb.GumbelCDF(1.0, 2.0)
>>> # To use a GumbelCDF bijector in a network.
>>> x = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> y = Tensor([0.1, 0.2, 0.3], dtype=mindspore.float32)
>>> ans1 = gumbel_cdf.forward(x)
>>> print(ans1.shape)
(3,)
>>> ans2 = gumbel_cdf.inverse(y)
>>> print(ans2.shape)
(3,)
>>> ans3 = gumbel_cdf.forward_log_jacobian(x)
>>> print(ans3.shape)
(3,)
>>> ans4 = gumbel_cdf.inverse_log_jacobian(y)
>>> print(ans4.shape)
(3,)
```

property loc
Return the loc parameter of the bijector.

Returns

Tensor, the loc parameter of the bijector.

property scale
Return the scale parameter of the bijector.

Returns

Tensor, the scale parameter of the bijector.

forward(*value*)
forward mapping, compute the value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value to compute.

forward_log_jacobian(*value*)
compute the log value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of forward mapping.

inverse(*value*)
Inverse mapping, compute the value after inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the value of output after mapping.

inverse_log_jacobian(*value*)
Compute the log value of the inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the log value of the inverse mapping.

20.1.4 mindspore.nn.probability.bijector.Invert

class mindspore.nn.probability.bijector.**Invert**(*bijector*, *name=''*)

Invert Bijector. Compute the inverse function of the input bijector. If the function of the forward mapping, namely the input of *bijector* below, is $Y = g(X)$, then the function of corresponding inverse mapping Bijector is $Y = h(X) = g^{-1}(X)$.

Parameters

- **bijector** (Bijector) –Base Bijector.
- **name** (*str*) –The name of the Bijector. Default: "" . When name is set to "", it is actually 'Invert' + Bijector.name.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.bijector as msb
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def __init__(self):
...         super(Net, self).__init__()
...         self.origin = msb.ScalarAffine(scale=2.0, shift=1.0)
...         self.invert = msb.Invert(self.origin)
...
...     def construct(self, x_):
...         return self.invert.forward(x_)
>>> forward = Net()
>>> x = np.array([2.0, 3.0, 4.0, 5.0]).astype(np.float32)
>>> ans = forward(Tensor(x, dtype=mindspore.float32))
>>> print(ans.shape)
(4,)
```

property bijector

Return base bijector.

forward(*x*)

Compute the inverse mapping of underlying Bijector, namely $Y = h(X) = g^{-1}(X)$.

Parameters

- **x** (Tensor) - the value of output after mapping by the underlying bijector.

Returns

Tensor, the inverse mapping of underlying Bijector.

forward_log_jacobian(*x*)

Compute the log value of the inverse mapping of underlying Bijector $\log dg^{-1}(x)/dx$.

Parameters

- **x** (Tensor) - the value of output after mapping by the underlying bijector.

Returns

Tensor, the log value of the inverse mapping of underlying Bijector.

inverse(*y*)

Compute the forward mapping of underlying Bijector, namely $Y = g(X)$.

Parameters

- **y** (Tensor) - the value to compute by the underlying bijector.

Returns

Tensor, the forward mapping of underlying Bijector.

inverse_log_jacobian(*y*)

Compute the log value of the forward mapping of underlying Bijector, namely $Y = \log dg(x)/dx$.

Parameters

- **y** (Tensor) - the value to compute by the underlying bijector.

Returns

Tensor, the log value of forward mapping of underlying Bijector.

20.1.5 mindspore.nn.probability.bijector.PowerTransform

class mindspore.nn.probability.bijector.**PowerTransform**(*power=0.*, *name='PowerTransform'*)

PowerTransform Bijector. This Bijector performs the operation:

$$Y = g(X) = (1 + X * c)^{1/c}, X >= -1/c$$

where c >= 0 is the power.

The power transform maps inputs from *[-1/c, inf]* to *[0, inf]*.

This Bijector is equivalent to the *mindspore.nn.probability.bijector.Exp* bijector when *c=0*.

Parameters

- **power** (*float*, *list*, *numpy.ndarray*, Tensor) –The scale factor. Default: 0 .
- **name** (*str*) –The name of the bijector. Default: 'PowerTransform' .

Note: The dtype of *power* must be float.

Raises

- **ValueError** –When *power* is less than 0 or is not known statically.
- **TypeError** –When the dtype of *power* is not float.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.bijector as msb
>>> from mindspore import Tensor
>>> # To initialize a PowerTransform bijector of power 0.5.
>>> powertransform = msb.PowerTransform(0.5)
>>> value = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> ans1 = powertransform.forward(value)
>>> print(ans1.shape)
(3,)
>>> ans2 = powertransform.inverse(value)
>>> print(ans2.shape)
(3,)
>>> ans3 = powertransform.forward_log_jacobian(value)
>>> print(ans3.shape)
(3,)
>>> ans4 = powertransform.inverse_log_jacobian(value)
>>> print(ans4.shape)
(3,)
```

property power

Return the power index.

Returns

Tensor, the power index.

forward(*value*)

forward mapping, compute the value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value to compute.

forward_log_jacobian(*value*)

compute the log value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of forward mapping.

inverse(*value*)

Inverse mapping, compute the value after inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the value of output after mapping.

inverse_log_jacobian(*value*)

Compute the log value of the inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the log value of the inverse mapping.

20.1.6 mindspore.nn.probability.bijector.ScalarAffine

class mindspore.nn.probability.bijector.**ScalarAffine**(*scale=1.0*, *shift=0.0*, *name='ScalarAffine'*)

Scalar Affine Bijector. This Bijector performs the operation:

$$Y = a * X + b$$

where a is the scale factor and b is the shift factor.

Parameters

- **scale** (*float*, *list*, *numpy.ndarray*, Tensor) –The scale factor. a in the formula. Default: 1.0.
- **shift** (*float*, *list*, *numpy.ndarray*, Tensor) –The shift factor. b in the formula. Default: 0.0.
- **name** (*str*) –The name of the bijector. Default: 'ScalarAffine'.

Note: The dtype of *shift* and *scale* must be float. If *shift*, *scale* are passed in as numpy.ndarray or tensor, they have to have the same dtype otherwise an error will be raised.

Raises

TypeError –When the dtype of *shift* or *scale* is not float, and when the dtype of *shift* and *scale* is not same.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>>
>>> # To initialize a ScalarAffine bijector of scale 1.0 and shift 2.
>>> scalaraffine = nn.probability.bijector.ScalarAffine(1.0, 2.0)
>>> value = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> ans1 = scalaraffine.forward(value)
>>> print(ans1.shape)
(3,)
>>> ans2 = scalaraffine.inverse(value)
>>> print(ans2.shape)
(3,)
>>> ans3 = scalaraffine.forward_log_jacobian(value)
>>> print(ans3.shape)
()
>>> ans4 = scalaraffine.inverse_log_jacobian(value)
>>> print(ans4.shape)
()
```

property shift
Return the shift parameter of the bijector.

Returns

Tensor, the shift parameter of the bijector.

property scale
Return the scale parameter of the bijector.

Returns

Tensor, the scale parameter of the bijector.

forward(*value*)
forward mapping, compute the value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value to compute.

forward_log_jacobian(*value*)
compute the log value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of forward mapping.

inverse(*value*)
Inverse mapping, compute the value after inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the value of input after mapping.

inverse_log_jacobian(*value*)
Compute the log value of the inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the log value of the inverse mapping.

20.1.7 mindspore.nn.probability.bijector.Softplus

class mindspore.nn.probability.bijector.**Softplus**(*sharpness=1.0*, *name='Softplus'*)

Softplus Bijector. This Bijector performs the operation:

$$Y = \frac{\log(1 + e^{kX})}{k}$$

where k is the sharpness factor.

Parameters

- **sharpness** (*float*, *list*, *numpy.ndarray*, Tensor) – The scale factor. k in the above formula. Default: 1.0.
- **name** (*str*) – The name of the Bijector. Default: 'Softplus'.

Note: The dtype of *sharpness* must be float.

Raises

TypeError – When the dtype of the sharpness is not float.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.bijector as msb
>>> from mindspore import Tensor
>>>
>>> # To initialize a Softplus bijector of sharpness 2.0.
>>> softplus = msb.Softplus(2.0)
>>> # To use a Softplus bijector in a network.
>>> value = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> ans1 = softplus.forward(value)
>>> print(ans1.shape)
(3,)
>>> ans2 = softplus.inverse(value)
>>> print(ans2.shape)
(3,)
>>> ans3 = softplus.forward_log_jacobian(value)
>>> print(ans3.shape)
(3,)
>>> ans4 = softplus.inverse_log_jacobian(value)
>>> print(ans4.shape)
(3,)
```

property sharpness

Return the sharpness parameter of the distribution.

Returns

Tensor, the sharpness parameter of the distribution.

forward(*value*)
forward mapping, compute the value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value to compute.

forward_log_jacobian(*value*)
compute the log value after mapping.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of forward mapping.

inverse(*value*)
Inverse mapping, compute the value after inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the value of output after mapping.

inverse_log_jacobian(*value*)
Compute the log value of the inverse mapping.

Parameters

- **value** (Tensor) - the value of output after mapping.

Returns

Tensor, the log value of the inverse mapping.

20.2 Distributions

API Name	Description	Supported Platforms
mindspore.nn.probability.distribution.Bernoulli	Bernoulli Distribution.	Ascend GPU
mindspore.nn.probability.distribution.Beta	Beta distribution.	Ascend
mindspore.nn.probability.distribution.Categorical	Categorical distribution.	Ascend GPU
mindspore.nn.probability.distribution.Cauchy	Cauchy distribution.	Ascend
mindspore.nn.probability.distribution.Distribution	Base class for all mathematical distributions.	Ascend GPU
mindspore.nn.probability.distribution.Exponential	Exponential Distribution.	Ascend GPU

continues on next page

Table 2 – continued from previous page

mindspore.nn.probability.distribution.Gamma	Gamma distribution.	Ascend
mindspore.nn.probability.distribution.Geometric	Geometric Distribution.	Ascend GPU
mindspore.nn.probability.distribution.Gumbel	Gumbel distribution.	Ascend GPU
mindspore.nn.probability.distribution.HalfNormal	HalfNormal distribution.	CPU
mindspore.nn.probability.distribution.Laplace	Laplace distribution.	Ascend GPU CPU
mindspore.nn.probability.distribution.Logistic	Logistic distribution.	Ascend GPU
mindspore.nn.probability.distribution.LogNormal	LogNormal distribution.	Ascend GPU
mindspore.nn.probability.distribution.Normal	Normal distribution.	Ascend GPU
mindspore.nn.probability.distribution.Poisson	Poisson Distribution.	Ascend
mindspore.nn.probability.distribution.StudentT	StudentT distribution.	CPU
mindspore.nn.probability.distribution.TransformedDistribution	Transformed Distribution.	Ascend GPU CPU
mindspore.nn.probability.distribution.Uniform	Uniform Distribution.	Ascend GPU CPU

20.2.1 mindspore.nn.probability.distribution.Bernoulli

class mindspore.nn.probability.distribution.**Bernoulli**(*probs=None*, *seed=None*, *dtype=mstype.int32*, *name='Bernoulli'*)

Bernoulli Distribution. A Bernoulli Distribution is a discrete distribution with the range $\{0, 1\}$ and the probability mass function as $P(X = 0) = p, P(X = 1) = 1 - p$.

Parameters

- **probs** (*float*, *list*, *numpy.ndarray*, *Tensor*) – The probability of that the outcome is 1. Default: `None`.
- **seed** (*int*) – The seed used in sampling. The global seed is used if it is None. Default: `None`.
- **dtype** (*mindspore.dtype*) – The type of the event samples. Default: `mstype.int32`.
- **name** (*str*) – The name of the distribution. Default: `'Bernoulli'`.

Note: *probs* must be a proper probability (0 < p < 1). *dist_spec_args* is *probs*.

Raises

ValueError – When p <= 0 or p >=1.

Supported Platforms:

`Ascend GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Bernoulli distribution of the probability 0.5.
>>> b1 = msd.Bernoulli(0.5, dtype=mindspore.int32)
>>> # A Bernoulli distribution can be initialized without arguments.
>>> # In this case, `probs` must be passed in through arguments during function calls.
>>> b2 = msd.Bernoulli(dtype=mindspore.int32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1, 0, 1], dtype=mindspore.int32)
>>> probs_a = Tensor([0.6], dtype=mindspore.float32)
>>> probs_b = Tensor([0.2, 0.3, 0.4], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`, are the␣
↪same as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     probs1 (Tensor): the probability of success. Default: self.probs.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing `prob` by the name of the function.
>>> ans = b1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate `prob` with respect to distribution b.
>>> ans = b1.prob(value, probs_b)
>>> print(ans.shape)
(3,)
>>> # `probs` must be passed in during function calls.
>>> ans = b2.prob(value, probs_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     probs1 (Tensor): the probability of success. Default: self.probs.
>>> # Examples of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = b1.mean() # return 0.5
>>> print(ans.shape)
()
>>> ans = b1.mean(probs_b) # return probs_b
>>> print(ans.shape)
(3,)
>>> # `probs` must be passed in during function calls.
>>> ans = b2.mean(probs_a)
>>> print(ans.shape)
(1,)
>>> # Interfaces of `kl_loss` and `cross_entropy` are the same as follows:
>>> # Args:
>>> #     dist (str): the name of the distribution. Only 'Bernoulli' is supported.
>>> #     probs1_b (Tensor): the probability of success of distribution b.
>>> #     probs1_a (Tensor): the probability of success of distribution a. Default: self.
↪probs.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
```

(continues on next page)

(continued from previous page)

```
>>> ans = b1.kl_loss('Bernoulli', probs_b)
>>> print(ans.shape)
(3,)
>>> ans = b1.kl_loss('Bernoulli', probs_b, probs_a)
>>> print(ans.shape)
(3,)
>>> # An additional `probs_a` must be passed in.
>>> ans = b2.kl_loss('Bernoulli', probs_b, probs_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ().
>>> #     probs1 (Tensor): the probability of success. Default: self.probs.
>>> ans = b1.sample()
>>> print(ans.shape)
()
>>> ans = b1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = b1.sample((2,3), probs_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = b2.sample((2,3), probs_a)
>>> print(ans.shape)
(2, 3, 1)
```

property probs

Return the probability of success, namely the output is 1.

Returns

Tensor, the probability of success.

cdf (*value*, *probs1*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *probs1_b*, *probs1_a*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the distribution.
- **probs1_b** (Tensor) - the probability of success of the distribution b.
- **probs1_a** (Tensor) - the probability of success of the distribution a. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*probs1=None*)
Compute the value of the entropy.

Parameters

- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *probs1_b*, *probs1_a*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **probs1_b** (Tensor) - the probability of success of the distribution a.
- **probs1_a** (Tensor) - the probability of success of the distribution b. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *probs1*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *probs1*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *probs1*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*probs1*)
Compute the mean value of the distribution.

Parameters

- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode (*probs1*)

Compute the mode value of the distribution.

Parameters

- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob (*value*, *probs1*)

The probability of the given value. For the discrete distribution, it is the probability mass function(pmf).

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the probability.

sample (*shape*, *probs1*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd (*probs1*)

The standard deviation.

Parameters

- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *probs1*)

Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*probs1*)

Compute the variance of the distribution.

Parameters

- **probs1** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.2 mindspore.nn.probability.distribution.Beta

class mindspore.nn.probability.distribution.**Beta**(*concentration1=None*, *concentration0=None*, *seed=None*, *dtype=mstype.float32*, *name='Beta'*)

Beta distribution. A Beta distributio is a continuous distribution with the range $[0, 1]$ and the probability density function:

$$f(x, \alpha, \beta) = x^{\alpha}(1-x)^{\beta-1}/B(\alpha, \beta)$$

Where B is the Beta function.

Parameters

- **concentration1** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The concentration1, also know as $alpha$ of the Beta distribution. Default: `None` .
- **concentration0** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The concentration0, also know as $beta$ of the Beta distribution. Default: `None` .
- **seed** (*int*) −The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (mindspore.dtype) −The type of the event samples. Default: `mstype.float32` .
- **name** (*str*) −The name of the distribution. Default: `'Beta'` .

Note:

- *concentration1* and *concentration0* must be greater than zero.
- *dist_spec_args* are *concentration1* and *concentration0*.
- *dtype* must be a float type because Beta distributions are continuous.

Raises

- **ValueError** −When concentration1 <= 0 or concentration0 >=1.
- **TypeError** −When the input *dtype* is not a float or a subclass of float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Beta distribution of the concentration1 3.0 and the concentration0 4.0.
>>> b1 = msd.Beta([3.0], [4.0], dtype=mindspore.float32)
>>> # A Beta distribution can be initialized without arguments.
>>> # In this case, `concentration1` and `concentration0` must be passed in through␣
↪arguments.
>>> b2 = msd.Beta(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([0.1, 0.5, 0.8], dtype=mindspore.float32)
>>> concentration1_a = Tensor([2.0], dtype=mindspore.float32)
>>> concentration0_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> concentration1_b = Tensor([1.0], dtype=mindspore.float32)
>>> concentration0_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
>>> # `prob` and `log_prob`, have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     concentration1 (Tensor): the concentration1 of the distribution. Default: self._
↪concentration1.
>>> #     concentration0 (Tensor): the concentration0 of the distribution. Default: self._
↪concentration0.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function
>>> ans = b1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = b1.prob(value, concentration1_b, concentration0_b)
>>> print(ans.shape)
(3,)
>>> # `concentration1` and `concentration0` must be passed in during function calls
>>> ans = b2.prob(value, concentration1_a, concentration0_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, `mode`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     concentration1 (Tensor): the concentration1 of the distribution. Default: self._
↪concentration1.
>>> #     concentration0 (Tensor): the concentration0 of the distribution. Default: self._
↪concentration0.
>>> # Example of `mean`, `sd`, `mode`, `var`, and `entropy` are similar.
>>> ans = b1.mean()
>>> print(ans.shape)
(1,)
>>> ans = b1.mean(concentration1_b, concentration0_b)
>>> print(ans.shape)
(3,)
>>> # `concentration1` and `concentration0` must be passed in during function calls.
>>> ans = b2.mean(concentration1_a, concentration0_a)
>>> print(ans.shape)
```

(continues on next page)

(continued from previous page)

```
(3,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same:
>>> # Args:
>>> #     dist (str): the type of the distributions. Only "Beta" is supported.
>>> #     concentration1_b (Tensor): the concentration1 of distribution b.
>>> #     concentration0_b (Tensor): the concentration0 of distribution b.
>>> #     concentration1_a (Tensor): the concentration1 of distribution a.
>>> #       Default: self._concentration1.
>>> #     concentration0_a (Tensor): the concentration0 of distribution a.
>>> #       Default: self._concentration0.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = b1.kl_loss('Beta', concentration1_b, concentration0_b)
>>> print(ans.shape)
(3,)
>>> ans = b1.kl_loss('Beta', concentration1_b, concentration0_b, concentration1_a,␣
↪concentration0_a)
>>> print(ans.shape)
(3,)
>>> # Additional `concentration1` and `concentration0` must be passed in.
>>> ans = b2.kl_loss('Beta', concentration1_b, concentration0_b, concentration1_a,␣
↪concentration0_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     concentration1 (Tensor): the concentration1 of the distribution. Default: self._
↪concentration1.
>>> #     concentration0 (Tensor): the concentration0 of the distribution. Default: self._
↪concentration0.
>>> ans = b1.sample()
>>> print(ans.shape)
(1,)
>>> ans = b1.sample((2,3))
>>> print(ans.shape)
(2, 3, 1)
>>> ans = b1.sample((2,3), concentration1_b, concentration0_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = b2.sample((2,3), concentration1_a, concentration0_a)
>>> print(ans.shape)
(2, 3, 3)
```

property concentration0

Return concentration0, aka the beta parameter of the Beta distribution.

Returns

Tensor, the value of concentration0.

property concentration1

Return concentration1, aka the alpha parameter of the Beta distribution.

Returns

Tensor, the value of concentration1.

cdf(*value*, *concentration1*, *concentration0*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *concentration1_b*, *concentration0_b*, *concentration1*, *concentration0*)
Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **concentration1_b** (Tensor) - the alpha parameter of the other Beta distribution b.
- **concentration0_b** (Tensor) - the beta parameter of the other Beta distribution b.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution a. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution a. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy (*concentration1*, *concentration0*)
Compute the value of the entropy.

Parameters

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss (*dist*, *concentration1_b*, *concentration0_b*, *concentration1*, *concentration0*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **concentration1_b** (Tensor) - the alpha parameter of the other Beta distribution b.
- **concentration0_b** (Tensor) - the beta parameter of the other Beta distribution b.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution a. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution a. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf (*value*, *concentration1*, *concentration0*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob (*value*, *concentration1*, *concentration0*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival (*value*, *concentration1*, *concentration0*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean (*concentration1*, *concentration0*)
Compute the mean value of the distribution.

Parameters

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode (*concentration1*, *concentration0*)
Compute the mode value of the distribution.

Parameters

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob (*value*, *concentration1*, *concentration0*)
The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *concentration1*, *concentration0*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*concentration1*, *concentration0*)
The standard deviation.

Parameters

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *concentration1*, *concentration0*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*concentration1*, *concentration0*)
Compute the variance of the distribution.

Parameters

- **concentration1** (Tensor) - the alpha parameter of the Beta distribution. Default: `None` .
- **concentration0** (Tensor) - the beta parameter of the Beta distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.3 mindspore.nn.probability.distribution.Categorical

class mindspore.nn.probability.distribution.**Categorical**(*probs=None*, *seed=None*, *dtype=mstype.int32*, *name='Categorical'*)

Categorical distribution. A Categorical Distribution is a discrete distribution with the range $\{1, 2, ..., k\}$ and the probability mass function as $P(X = i) = p_i, i = 1, ..., k$.

Parameters

- **probs** (*Tensor, list, numpy.ndarray*) – Event probabilities. Default: `None`.
- **seed** (*int*) – The global seed is used in sampling. Global seed is used if it is None. Default: `None`.
- **dtype** (*mindspore.dtype*) – The type of the event samples. Default: `mstype.int32`.
- **name** (*str*) – The name of the distribution. Default: `'Categorical'`.

Note: *probs* must have rank at least 1, values are proper probabilities and sum to 1.

Raises

ValueError – When the sum of all elements in *probs* is not 1.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Categorical distribution of probs [0.5, 0.5]
>>> ca1 = msd.Categorical(probs=[0.2, 0.8], dtype=mindspore.int32)
>>> # A Categorical distribution can be initialized without arguments.
>>> # In this case, `probs` must be passed in through arguments during function calls.
>>> ca2 = msd.Categorical(dtype=mindspore.int32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1, 0], dtype=mindspore.int32)
>>> probs_a = Tensor([0.5, 0.5], dtype=mindspore.float32)
>>> probs_b = Tensor([0.35, 0.65], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`, are the␣
↪same as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     probs (Tensor): event probabilities. Default: self.probs.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing `prob` by the name of the function.
>>> ans = ca1.prob(value)
>>> print(ans.shape)
(2,)
>>> # Evaluate `prob` with respect to distribution b.
```

(continues on next page)

(continued from previous page)

```
>>> ans = ca1.prob(value, probs_b)
>>> print(ans.shape)
(2,)
>>> # `probs` must be passed in during function calls.
>>> ans = ca2.prob(value, probs_a)
>>> print(ans.shape)
(2,)
>>> # Functions `mean`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     probs (Tensor): event probabilities. Default: self.probs.
>>> # Examples of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = ca1.mean() # return 0.8
>>> print(ans.shape)
(1,)
>>> ans = ca1.mean(probs_b)
>>> print(ans.shape)
(1,)
>>> # `probs` must be passed in during function calls.
>>> ans = ca2.mean(probs_a)
>>> print(ans.shape)
(1,)
>>> # Interfaces of `kl_loss` and `cross_entropy` are the same as follows:
>>> # Args:
>>> #     dist (str): the name of the distribution. Only 'Categorical' is supported.
>>> #     probs_b (Tensor): event probabilities of distribution b.
>>> #     probs (Tensor): event probabilities of distribution a. Default: self.probs.
>>> # Examples of `kl_loss`, `cross_entropy` is similar.
>>> ans = ca1.kl_loss('Categorical', probs_b)
>>> print(ans.shape)
()
>>> ans = ca1.kl_loss('Categorical', probs_b, probs_a)
>>> print(ans.shape)
()
>>> # An additional `probs` must be passed in.
>>> ans = ca2.kl_loss('Categorical', probs_b, probs_a)
>>> print(ans.shape)
()
```

property probs

Return the event probability.

Returns

Tensor, the event probability.

cdf (*value*, *probs*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *probs_b*, *probs*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **probs_b** (Tensor) - the event probability of the distribution b.
- **probs** (Tensor) - the event probability of the distribution a. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*probs*)

Compute the value of the entropy.

Parameters

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *probs_b*, *probs*)

Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **probs_b** (Tensor) - the event probability of the distribution b.
- **probs** (Tensor) - the event probability of the distribution a. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *probs*)

Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *probs*)

the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *probs*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*probs*)

Compute the mean value of the distribution.

Parameters

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*probs*)

Compute the mode value of the distribution.

Parameters

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *probs*)

The probability of the given value. For the discrete distribution, it is the probability mass function(pmf).

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *probs*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*probs*)

The standard deviation.

Parameters

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *probs*)

Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*probs*)

Compute the variance of the distribution.

Parameters

- **probs** (Tensor) - the event probability. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.4 mindspore.nn.probability.distribution.Cauchy

class mindspore.nn.probability.distribution.**Cauchy**(*loc=None*, *scale=None*, *seed=None*, *dtype=mstype.float32*, *name='Cauchy'*)

Cauchy distribution. A Cauchy distributio is a continuous distribution with the range of all real numbers and the probability density function:

$$f(x, a, b) = 1/\pi b(1 - ((x - a)/b)^2)$$

Where a, b are loc and scale parameter respectively.

Parameters

- **loc** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The location of the Cauchy distribution. a in the formula. Default: `None` .
- **scale** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The scale of the Cauchy distribution. b in the formula. Default: `None` .
- **seed** (*int*) −The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (mindspore.dtype) −The type of the event samples. Default: `mstype.float32` .
- **name** (*str*) −The name of the distribution. Default: `'Cauchy'` .

Note: *scale* must be greater than zero. *dist_spec_args* are *loc* and *scale*. *dtype* must be a float type because Cauchy distributions are continuous. Cauchy distribution is not supported on GPU backend.

Raises

- **ValueError** −When scale <= 0.
- **TypeError** −When the input *dtype* is not a subclass of float.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Cauchy distribution of loc 3.0 and scale 4.0.
>>> cauchy1 = msd.Cauchy(3.0, 4.0, dtype=mindspore.float32)
>>> # A Cauchy distribution can be initialized without arguments.
>>> # In this case, 'loc' and `scale` must be passed in through arguments.
>>> cauchy2 = msd.Cauchy(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> loc_a = Tensor([2.0], dtype=mindspore.float32)
>>> scale_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> loc_b = Tensor([1.0], dtype=mindspore.float32)
>>> scale_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`,
>>> # have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function
>>> ans = cauchy1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to distribution b.
>>> ans = cauchy1.prob(value, loc_b, scale_b)
>>> print(ans.shape)
(3,)
>>> # `loc` and `scale` must be passed in during function calls
>>> ans = cauchy2.prob(value, loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mode` and `entropy` have the same arguments.
>>> # Args:
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> # Example of `mode`.
>>> ans = cauchy1.mode() # return 3.0
>>> print(ans.shape)
()
>>> ans = cauchy1.mode(loc_b, scale_b) # return loc_b
>>> print(ans.shape)
(3,)
>>> # `loc` and `scale` must be passed in during function calls.
>>> ans = cauchy2.mode(loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same:
>>> # Args:
>>> #     dist (str): the type of the distributions. Only "Cauchy" is supported.
```

(continues on next page)

(continued from previous page)

```
>>> #     loc_b (Tensor): the loc of distribution b.
>>> #     scale_b (Tensor): the scale distribution b.
>>> #     loc (Tensor): the loc of distribution a. Default: self.loc.
>>> #     scale (Tensor): the scale distribution a. Default: self.scale.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = cauchy1.kl_loss('Cauchy', loc_b, scale_b)
>>> print(ans.shape)
(3,)
>>> ans = cauchy1.kl_loss('Cauchy', loc_b, scale_b, loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Additional `loc` and `scale` must be passed in.
>>> ans = cauchy2.kl_loss('Cauchy', loc_b, scale_b, loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> ans = cauchy1.sample()
>>> print(ans.shape)
()
>>> ans = cauchy1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = cauchy1.sample((2,3), loc_b, scale_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = cauchy2.sample((2,3), loc_a, scale_a)
>>> print(ans.shape)
(2, 3, 3)
```

property loc

Return the loc parameter of the distribution.

Returns

Tensor, the loc parameter of the distribution.

property scale

Return the scale parameter of the distribution.

Returns

Tensor, the scale parameter of the distribution.

cdf(*value*, *loc*, *scale*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy(*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*loc*, *scale*)
Compute the value of the entropy.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *loc*, *scale*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *loc*, *scale*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *loc*, *scale*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*loc*, *scale*)

Compute the mean value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*loc*, *scale*)

Compute the mode value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *loc*, *scale*)

The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *loc*, *scale*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd (*loc*, *scale*)
The standard deviation.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *loc*, *scale*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*loc*, *scale*)
Compute the variance of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.5 mindspore.nn.probability.distribution.Distribution

class mindspore.nn.probability.distribution.**Distribution** (*seed*, *dtype*, *name*, *param*)
Base class for all mathematical distributions.

Parameters

- **seed** (*int*) −The seed is used in sampling. 0 is used if it is None.
- **dtype** (mindspore.dtype) −The type of the event samples.
- **name** (*str*) −The name of the distribution.
- **param** (*dict*) −The parameters used to initialize the distribution.

Note: Derived class must override operations such as *_mean*, *_prob*, and *_log_prob*. Required arguments, such as *value* for *_prob*, must be passed in through *args* or *kwargs*. *dist_spec_args* which specifies a new distribution are optional.

dist_spec_args is unique for each type of distribution. For example, *mean* and *sd* are the *dist_spec_args* for a Normal distribution, while *rate* is the *dist_spec_args* for an Exponential distribution.

For all functions, passing in *dist_spec_args*, is optional. Function calls with the additional *dist_spec_args* passed in will evaluate the result with a new distribution specified by the *dist_spec_args*. However, it will not change the original distribution.

Supported Platforms:

`Ascend` `GPU`

cdf(*value*, **args*, ***kwargs*)

Evaluate the cumulative distribution function(cdf) at given value.

Parameters

- **value** (Tensor) −value to be evaluated.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the cdf of the distribution.

construct(*name*, **args*, ***kwargs*)

Override *construct* in Cell.

Note: Names of supported functions include: 'prob', 'log_prob', 'cdf', 'log_cdf', 'survival_function', 'log_survival', 'var', 'sd', 'mode', 'mean', 'entropy', 'kl_loss', 'cross_entropy', 'sample', 'get_dist_args', and 'get_dist_type'.

Parameters

- **name** (*str*) −The name of the function.
- ***args** (*list*) −A list of positional arguments that the function needs.
- ****kwargs** (*dict*) −A dictionary of keyword arguments that the function needs.

Returns

Tensor, the value of corresponding computation method.

cross_entropy(*dist*, **args*, ***kwargs*)

Evaluate the cross_entropy between distribution a and b.

Parameters

- **dist** (*str*) −type of the distribution.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.

- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: *dist_spec_args* of distribution b must be passed to the function through *args* or *kwargs*. Passing in *dist_spec_args* of distribution a is optional.

Returns

Tensor, the cross_entropy of two distributions.

entropy(**args*, ***kwargs*)
Evaluate the entropy.

Parameters

- ***args** (*list*) –the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the entropy of the distribution.

get_dist_args(**args*, ***kwargs*)
Check the availability and validity of default parameters and *dist_spec_args*.

Parameters

- ***args** (*list*) –the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: *dist_spec_args* must be passed in through list or dictionary. The order of *dist_spec_args* should follow the initialization order of default parameters through *_add_parameter*. If some *dist_spec_args* is None, the corresponding default parameter is returned.

Returns

list[Tensor], the list of parameters.

get_dist_type()
Return the type of the distribution.

Returns

string, the name of distribution.

kl_loss(*dist*, **args*, ***kwargs*)
Evaluate the KL divergence, i.e. KL(a||b).

Parameters

- **dist** (*str*) –type of the distribution.
- ***args** (*list*) –the list of positional arguments forwarded to subclasses.

- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: *dist_spec_args* of distribution b must be passed to the function through *args* or *kwargs*. Passing in *dist_spec_args* of distribution a is optional.

Returns

Tensor, the kl loss function of the distribution.

log_cdf(*value*, **args*, ***kwargs*)
Evaluate the log the cumulative distribution function(cdf) at given value.

Parameters

- **value** (Tensor) –value to be evaluated.
- ***args** (*list*) –the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the log cdf of the distribution.

log_prob(*value*, **args*, ***kwargs*)
Evaluate the log probability(pdf or pmf) at the given value.

Parameters

- **value** (Tensor) –value to be evaluated.
- ***args** (*list*) –the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the value of log probability.

log_survival(*value*, **args*, ***kwargs*)
Evaluate the log survival function at given value.

Parameters

- **value** (Tensor) –value to be evaluated.
- ***args** (*list*) –the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) –the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the log survival function of the distribution.

mean(**args*, ***kwargs*)
Evaluate the mean.

Parameters

- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the mean of the distribution.

mode(**args*, ***kwargs*)
Evaluate the mode.

Parameters

- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the mode of the distribution.

prob(*value*, **args*, ***kwargs*)
Evaluate the probability (pdf or pmf) at given value. For a discrete distribution, it is a probability mass function, while for a continuous distribution, it is probability density function.

Parameters

- **value** (Tensor) −value to be evaluated.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the value of probability.

sample(**args*, ***kwargs*)
Sampling function.

Parameters

- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the sample generated from the distribution.

sd(**args*, ***kwargs*)
Evaluate the standard deviation.

Parameters

- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, **args*, ***kwargs*)
Evaluate the survival function at given value.

Parameters

- **value** (Tensor) −value to be evaluated.
- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the survival function of the distribution.

var(**args*, ***kwargs*)
Evaluate the variance.

Parameters

- ***args** (*list*) −the list of positional arguments forwarded to subclasses.
- ****kwargs** (*dict*) −the dictionary of keyword arguments forwarded to subclasses.

Note: A distribution can be optionally passed to the function by passing its *dist_spec_args* through *args* or *kwargs*.

Returns

Tensor, the variance of the distribution.

20.2.6 mindspore.nn.probability.distribution.Exponential

class mindspore.nn.probability.distribution.**Exponential**(*rate=None*, *seed=None*, *dtype=mstype.float32*, *name='Exponential'*)

Exponential Distribution. An Exponential distributio is a continuous distribution with the range $[0, \inf)$ and the probability density function:

$$f(x, \lambda) = \lambda \exp(-\lambda x)$$

where λ is the rate of the distribution.

Parameters

- **rate** (*int*, *float*, *list*, *numpy.ndarray*, *Tensor*, *optional*) –The inverse scale. λ in the formula. Default: `None` .
- **seed** (*int*, *optional*) –The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (mindspore.dtype, *optional*) –The type of the event samples. Default: `mstype.float32` .
- **name** (*str*, *optional*) –The name of the distribution. Default: `'Exponential'` .

Note:

- *rate* must be strictly greater than 0.
- *dtype* must be a float type because Exponential distributions are continuous.

Raises

- **ValueError** –When rate <= 0.
- **TypeError** –When the input *dtype* is not a subclass of float.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Exponential distribution of the probability 0.5.
>>> e1 = msd.Exponential(0.5, dtype=mindspore.float32)
>>> # An Exponential distribution can be initialized without arguments.
>>> # In this case, `rate` must be passed in through `args` during function calls.
>>> e2 = msd.Exponential(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1, 2, 3], dtype=mindspore.float32)
>>> rate_a = Tensor([0.6], dtype=mindspore.float32)
>>> rate_b = Tensor([0.2, 0.5, 0.4], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
```

(continues on next page)

(continued from previous page)

```
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`, are the␣
↪same as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     rate (Tensor): the rate of the distribution. Default: self.rate.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing `prob` by the name of the function.
>>> ans = e1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to distribution b.
>>> ans = e1.prob(value, rate_b)
>>> print(ans.shape)
(3,)
>>> # `rate` must be passed in during function calls.
>>> ans = e2.prob(value, rate_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, 'var', and 'entropy' have the same arguments as follows.
>>> # Args:
>>> #     rate (Tensor): the rate of the distribution. Default: self.rate.
>>> # Examples of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = e1.mean() # return 2
>>> print(ans.shape)
()
>>> ans = e1.mean(rate_b) # return 1 / rate_b
>>> print(ans.shape)
(3,)
>>> # `rate` must be passed in during function calls.
>>> ans = e2.mean(rate_a)
>>> print(ans.shape)
(1,)
>>> # Interfaces of `kl_loss` and `cross_entropy` are the same.
>>> # Args:
>>> #     dist (str): The name of the distribution. Only 'Exponential' is supported.
>>> #     rate_b (Tensor): the rate of distribution b.
>>> #     rate_a (Tensor): the rate of distribution a. Default: self.rate.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = e1.kl_loss('Exponential', rate_b)
>>> print(ans.shape)
(3,)
>>> ans = e1.kl_loss('Exponential', rate_b, rate_a)
>>> print(ans.shape)
(3,)
>>> # An additional `rate` must be passed in.
>>> ans = e2.kl_loss('Exponential', rate_b, rate_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     probs1 (Tensor): the rate of the distribution. Default: self.rate.
>>> ans = e1.sample()
>>> print(ans.shape)
()
>>> ans = e1.sample((2,3))
```

(continues on next page)

(continued from previous page)

```
>>> print(ans.shape)
(2, 3)
>>> ans = e1.sample((2,3), rate_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = e2.sample((2,3), rate_a)
>>> print(ans.shape)
(2, 3, 1)
```

property rate

Return rate.

Returns

Tensor, the rate of the distribution.

cdf (*value*, *rate=None*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *rate_b*, *rate=None*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **rate_b** (Tensor) - the rate b of the other distribution.
- **rate** (Tensor, optional) - the rate a of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy (*rate=None*)

Compute the value of the entropy.

Parameters

- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss (*dist*, *rate_b*, *rate=None*)

Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **rate_b** (Tensor) - the rate b of the other distribution.
- **rate** (Tensor, optional) - the rate a of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *rate=None*)

Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *rate=None*)

the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *rate=None*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*rate=None*)

Compute the mean value of the distribution.

Parameters

- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*rate=None*)

Compute the mode value of the distribution.

Parameters

- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *rate=None*)

The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *rate=None*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*rate=None*)
The standard deviation.

Parameters

- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *rate=None*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*rate=None*)
Compute the variance of the distribution.

Parameters

- **rate** (Tensor, optional) - the rate of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.7 mindspore.nn.probability.distribution.Gamma

class mindspore.nn.probability.distribution.**Gamma**(*concentration=None*, *rate=None*, *seed=None*, *dtype=mstype.float32*, *name='Gamma'*)

Gamma distribution. A Gamma distributio is a continuous distribution with the range $(0, \inf)$ and the probability density function:

$$f(x, \alpha, \beta) = \beta^{\alpha}/\Gamma(\alpha)x^{\alpha-1}\exp(-\beta x).$$

where G is the Gamma function, and α and β are the concentration and the rate of the distribution respectively.

Parameters

- **concentration** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The concentration, also know as α of the Gamma distribution. Default: None .
- **rate** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The rate, also know as β of the Gamma distribution. Default: None .
- **seed** (*int*) –The seed used in sampling. The global seed is used if it is None. Default: None .
- **dtype** (mindspore.dtype) –The type of the event samples. Default: mstype.float32 .
- **name** (*str*) –The name of the distribution. Default: 'Gamma' .

Note: *concentration* and *rate* must be greater than zero. *dist_spec_args* are *concentration* and *rate*. *dtype* must be a float type because Gamma distributions are continuous.

Raises

- **ValueError** –When concentration <= 0 or rate <= 0.
- **TypeError** –When the input *dtype* is not a subclass of float.

Supported Platforms:

Ascend

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Gamma distribution of the concentration 3.0 and the rate 4.0.
>>> g1 = msd.Gamma([3.0], [4.0], dtype=mindspore.float32)
>>> # A Gamma distribution can be initialized without arguments.
>>> # In this case, `concentration` and `rate` must be passed in through arguments.
>>> g2 = msd.Gamma(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> concentration_a = Tensor([2.0], dtype=mindspore.float32)
>>> rate_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> concentration_b = Tensor([1.0], dtype=mindspore.float32)
>>> rate_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>>
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
```

(continues on next page)

(continued from previous page)

```
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`,
>>> # have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     concentration (Tensor): the concentration of the distribution. Default: self._
 ↪concentration.
>>> #     rate (Tensor): the rate of the distribution. Default: self._rate.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function
>>> ans = g1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = g1.prob(value, concentration_b, rate_b)
>>> print(ans.shape)
(3,)
>>> # `concentration` and `rate` must be passed in during function calls for g2.
>>> ans = g2.prob(value, concentration_a, rate_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, `mode`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     concentration (Tensor): the concentration of the distribution. Default: self._
 ↪concentration.
>>> #     rate (Tensor): the rate of the distribution. Default: self._rate.
>>> # Example of `mean`, `sd`, `mode`, `var`, and `entropy` are similar.
>>> ans = g1.mean()
>>> print(ans.shape)
(1,)
>>> ans = g1.mean(concentration_b, rate_b)
>>> print(ans.shape)
(3,)
>>> # `concentration` and `rate` must be passed in during function calls.
>>> ans = g2.mean(concentration_a, rate_a)
>>> print(ans.shape)
(3,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same:
>>> # Args:
>>> #     dist (str): the type of the distributions. Only "Gamma" is supported.
>>> #     concentration_b (Tensor): the concentration of distribution b.
>>> #     rate_b (Tensor): the rate of distribution b.
>>> #     concentration_a (Tensor): the concentration of distribution a. Default: self._
 ↪concentration.
>>> #     rate_a (Tensor): the rate of distribution a. Default: self._rate.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = g1.kl_loss('Gamma', concentration_b, rate_b)
>>> print(ans.shape)
(3,)
>>> ans = g1.kl_loss('Gamma', concentration_b, rate_b, concentration_a, rate_a)
>>> print(ans.shape)
(3,)
>>> # Additional `concentration` and `rate` must be passed in.
>>> ans = g2.kl_loss('Gamma', concentration_b, rate_b, concentration_a, rate_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
```

(continues on next page)

(continued from previous page)

```
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     concentration (Tensor): the concentration of the distribution. Default: self._
 ↪concentration.
>>> #     rate (Tensor): the rate of the distribution. Default: self._rate.
>>> ans = g1.sample()
>>> print(ans.shape)
(1,)
>>> ans = g1.sample((2,3))
>>> print(ans.shape)
(2, 3, 1)
>>> ans = g1.sample((2,3), concentration_b, rate_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = g2.sample((2,3), concentration_a, rate_a)
>>> print(ans.shape)
(2, 3, 3)
```

property concentration

Return the concentration, aka the α parameter, of the distribution.

Returns

Tensor, concentration.

property rate

Return the rate, aka the β parameter, of the distribution.

Returns

Tensor, rate.

cdf(*value*, *concentration*, *rate*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy(*dist*, *concentration_b*, *rate_b*, *concentration*, *rate*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **concentration_b** (Tensor) - the α parameter of the other distribution.
- **rate_b** (Tensor) - the β parameter of the other distribution.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*concentration*, *rate*)
Compute the value of the entropy.

Parameters

- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *concentration_b*, *rate_b*, *concentration*, *rate*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **concentration_b** (Tensor) - the α parameter of the other distribution.
- **rate_b** (Tensor) - the β parameter of the other distribution.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *concentration*, *rate*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *concentration*, *rate*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *concentration*, *rate*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .

- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*concentration*, *rate*)
Compute the mean value of the distribution.

Parameters

- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*concentration*, *rate*)
Compute the mode value of the distribution.

Parameters

- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *concentration*, *rate*)
The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *concentration*, *rate*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*concentration*, *rate*)
The standard deviation.

Parameters

- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *concentration*, *rate*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*concentration*, *rate*)
Compute the variance of the distribution.

Parameters

- **concentration** (Tensor) - the α parameter of the distribution. Default: `None` .
- **rate** (Tensor) - the β parameter of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.8 mindspore.nn.probability.distribution.Geometric

class mindspore.nn.probability.distribution.**Geometric** (*probs=None*, *seed=None*, *dtype=mstype.int32*, *name='Geometric'*)

Geometric Distribution. A Geometric Distribution is a discrete distribution with the range as the non-negative integers, and the probability mass function as $P(X = i) = p(1 - p)^{i-1}, i = 1, 2, ...$ It represents that there are k failures before the first success, namely that there are in total k+1 Bernoulli trials when the first success is achieved.

Parameters

- **probs** (*float*, *list*, *numpy.ndarray*, `Tensor`) −The probability of success. Default: `None` .
- **seed** (*int*) −The seed used in sampling. Global seed is used if it is None. Default: `None` .
- **dtype** (`mindspore.dtype`) −The type of the event samples. Default: `mstype.int32` .
- **name** (*str*) −The name of the distribution. Default: `'Geometric'` .

Note: *probs* must be a proper probability (0 < p < 1). *dist_spec_args* is *probs*.

Raises

ValueError −When p <= 0 or p >= 1.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Geometric distribution of the probability 0.5.
>>> g1 = msd.Geometric(0.5, dtype=mindspore.int32)
>>> # A Geometric distribution can be initialized without arguments.
>>> # In this case, `probs` must be passed in through arguments during function calls.
>>> g2 = msd.Geometric(dtype=mindspore.int32)
>>>
>>> # Here are some tensors used below for testing
>>> value = Tensor([1, 0, 1], dtype=mindspore.int32)
>>> probs_a = Tensor([0.6], dtype=mindspore.float32)
>>> probs_b = Tensor([0.2, 0.5, 0.4], dtype=mindspore.float32)
>>>
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
 ↪including
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`,
>>> # have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     probs1 (Tensor): the probability of success of a Bernoulli trial. Default: self.
 ↪probs.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing `prob` by the name of the function.
>>> ans = g1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to distribution b.
>>> ans = g1.prob(value, probs_b)
>>> print(ans.shape)
(3,)
>>> # `probs` must be passed in during function calls.
>>> ans = g2.prob(value, probs_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     probs1 (Tensor): the probability of success of a Bernoulli trial. Default: self.
 ↪probs.
>>> # Examples of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = g1.mean() # return 1.0
>>> print(ans.shape)
()
>>> ans = g1.mean(probs_b)
>>> print(ans.shape)
(3,)
>>> # Probs must be passed in during function calls
>>> ans = g2.mean(probs_a)
>>> print(ans.shape)
(1,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same.
>>> # Args:
>>> #     dist (str): the name of the distribution. Only 'Geometric' is supported.
```

(continues on next page)

(continued from previous page)

```
>>> #     probs1_b (Tensor): the probability of success of a Bernoulli trial of distribution␣
 ↪b.
>>> #     probs1_a (Tensor): the probability of success of a Bernoulli trial of distribution␣
 ↪a.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = g1.kl_loss('Geometric', probs_b)
>>> print(ans.shape)
(3,)
>>> ans = g1.kl_loss('Geometric', probs_b, probs_a)
>>> print(ans.shape)
(3,)
>>> # An additional `probs` must be passed in.
>>> ans = g2.kl_loss('Geometric', probs_b, probs_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     probs1 (Tensor): the probability of success of a Bernoulli trial. Default: self.
 ↪probs.
>>> ans = g1.sample()
>>> print(ans.shape)
()
>>> ans = g1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = g1.sample((2,3), probs_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = g2.sample((2,3), probs_a)
>>> print(ans.shape)
(2, 3, 1)
```

property probs

Return the probability of success.

Returns

Tensor, the probability of success.

cdf (*value*, *probs*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *probs_b*, *probs*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **probs_b** (Tensor) - the probability of success of the other distribution.

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*probs*)

Compute the value of the entropy.

Parameters

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *probs_b*, *probs*)

Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **probs_b** (Tensor) - the probability of success of the other distribution.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *probs*)

Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *probs*)

the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *probs*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean (*probs*)
Compute the mean value of the distribution.

Parameters

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode (*probs*)
Compute the mode value of the distribution.

Parameters

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob (*value*, *probs*)
The probability of the given value. For the discrete distribution, it is the probability mass function(pmf).

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the probability.

sample (*shape*, *probs*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd (*probs*)
The standard deviation.

Parameters

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *probs*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*probs*)

Compute the variance of the distribution.

Parameters

- **probs** (Tensor) - the probability of success. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.9 mindspore.nn.probability.distribution.Gumbel

class mindspore.nn.probability.distribution.**Gumbel**(*loc*, *scale*, *seed=0*, *dtype=mstype.float32*, *name='Gumbel'*)

Gumbel distribution. A Gumbel distributio is a continuous distribution with the range of all real numbers and the probability density function:

$$f(x, a, b) = 1/b \exp(\exp(-(x - a)/b) - x)$$

Where a, b are loc and scale parameter respectively.

Parameters

- **loc** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The location of Gumbel distribution. a in the formula.
- **scale** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) −The scale of Gumbel distribution. b in the formula.
- **seed** (*int*) −the seed used in sampling. The global seed is used if it is None. Default: `0` .
- **dtype** (mindspore.dtype) −type of the distribution. Default: `mstype.float32` .
- **name** (*str*) −the name of the distribution. Default: `'Gumbel'` .

Note: *scale* must be greater than zero. *dist_spec_args* are *loc* and *scale*. *dtype* must be a float type because Gumbel distributions are continuous.

Raises

- **ValueError** −When scale <= 0.
- **TypeError** −When the input *dtype* is not a subclass of float.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import numpy as np
>>> import mindspore.nn.probability.distribution as msd
>>> import mindspore.nn as nn
>>> from mindspore import Tensor
>>> class Prob(nn.Cell):
...     def __init__(self):
...         super(Prob, self).__init__()
...         self.gum = msd.Gumbel(np.array([0.0]), np.array([[1.0], [2.0]]), dtype=mindspore.
↪float32)
...
...     def construct(self, x_):
...         return self.gum.prob(x_)
>>> value = np.array([1.0, 2.0]).astype(np.float32)
>>> pdf = Prob()
>>> output = pdf(Tensor(value, dtype=mindspore.float32))
```

property loc

Return the loc parameter of the distribution.

Returns

Tensor, the loc parameter of the distribution.

property scale

Return the scale parameter of the distribution.

Returns

Tensor, the scale parameter of the distribution.

cdf (*value*, *loc*, *scale*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *loc_b*, *scale_b*, *loc*, *scale*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*loc*, *scale*)
Compute the value of the entropy.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *loc*, *scale*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *loc*, *scale*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *loc*, *scale*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*loc*, *scale*)

Compute the mean value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*loc*, *scale*)

Compute the mode value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *loc*, *scale*)

The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *loc*, *scale*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*loc*, *scale*)

The standard deviation.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .

- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *loc*, *scale*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*loc*, *scale*)
Compute the variance of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.10 mindspore.nn.probability.distribution.HalfNormal

class mindspore.nn.probability.distribution.**HalfNormal** (*mean=None*, *sd=None*, *seed=None*, *dtype=mstype.float32*, *name='HalfNormal'*)

HalfNormal distribution. A HalfNormal distribution is a continuous distribution with the range $[\mu, \inf)$ and the probability density function:

$$f(x, \mu, \sigma) = 1/\sigma\sqrt{2\pi}\exp(-(x-\mu)^2/2\sigma^2).$$

where μ, σ are the mean and the standard deviation of the half normal distribution respectively.

Parameters

- **mean** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) – The mean of the distribution. μ in the formula. If this arg is `None` , then the mean of the distribution will be passed in runtime. Default: `None` .
- **sd** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) – The standard deviation of the distribution. σ in the formula. If this arg is `None` , then the sd of the distribution will be passed in runtime. Default: `None` .
- **seed** (*int, optional*) – The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (mindspore.dtype, *optional*) – The type of the event samples. Default: `mstype.float32` .
- **name** (*str, optional*) – The name of the distribution. Default: `'HalfNormal'` .

Note:

- *sd* must be greater than zero.

- *dtype* must be a float type because HalfNormal distributions are continuous.
- If the arg *mean* or *sd* is passed in runtime, then it will be used as the parameter value. Otherwise, the value passed in the constructor will be used.

Raises

- **ValueError** −When sd <= 0.
- **TypeError** −When the input *dtype* is not a float or a subclass of float.

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore.nn.probability.distribution import HalfNormal
>>> from mindspore import Tensor
>>> # To initialize a HalfNormal distribution of the mean 3.0 and the standard deviation 4.0.
>>> n1 = HalfNormal(3.0, 4.0, dtype=mindspore.float32)
>>> # A HalfNormal distribution can be initialized without arguments.
>>> # In this case, `mean` and `sd` must be passed in through arguments.
>>> hn = HalfNormal(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> mean_a = Tensor([2.0], dtype=mindspore.float32)
>>> sd_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> mean_b = Tensor([1.0], dtype=mindspore.float32)
>>> sd_b = Tensor([1.0, 1.5, 2.5], dtype=mindspore.float32)
>>> ans = n1.log_prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = n1.log_prob(value, mean_b, sd_b)
>>> print(ans.shape)
(3,)
>>> # `mean` and `sd` must be passed in during function calls
>>> ans = hn.log_prob(value, mean_a, sd_a)
>>> print(ans.shape)
(3,)
```

log_prob(*value*, *mean=None*, *sd=None*)

Evaluate log probability of the value of the HalfNormal distribution.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor, optional) - the mean of the distribution. Default: `None` .
- **sd** (Tensor, optional) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

20.2.11 mindspore.nn.probability.distribution.Laplace

class mindspore.nn.probability.distribution.**Laplace**(*mean=None*, *sd=None*, *seed=None*, *dtype=mstype.float32*, *name='Laplace'*)

Laplace distribution. A Laplace distribution is a continuous distribution with the range $(-\inf, \inf)$ and the probability density function:

$$f(x, \mu, b) = 1/(2 * b) * \exp(-abs(x - \mu)/b).$$

where μ, b are the mean and the scale of the laplace distribution respectively.

Parameters

- **mean** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) – The mean of the distribution. If this arg is `None` , then the mean of the distribution will be passed in runtime. Default: `None` .
- **sd** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) – The scale of the distribution. If this arg is `None` , then the scale of the distribution will be passed in runtime. Default: `None` .
- **seed** (*int, optional*) – The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (*mindspore.dtype, optional*) – The type of the event samples. Default: `mstype.float32` .
- **name** (*str, optional*) – The name of the distribution. Default: `'Laplace'` .

Note:

- *sd* must be greater than zero.
- *dtype* must be a float type because Laplace distributions are continuous.
- If the arg *mean* or *sd* is passed in runtime, then it will be used as the parameter value. Otherwise, the value passed in the constructor will be used.

Raises

- **ValueError** – When sd <= 0.
- **TypeError** – When the input *dtype* is not a subclass of float.

Supported Platforms:

`Ascend GPU CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> from mindspore.nn.probability.distribution import Laplace
>>> from mindspore import Tensor
>>> # To initialize a Laplace distribution of the mean 3.0 and the scale 4.0.
>>> n1 = Laplace(3.0, 4.0, dtype=mindspore.float32)
>>> # A Laplace distribution can be initialized without arguments.
>>> # In this case, `mean` and `sd` must be passed in through arguments.
>>> n2 = Laplace(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> mean_a = Tensor([2.0], dtype=mindspore.float32)
```

(continues on next page)

(continued from previous page)

```
>>> sd_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> mean_b = Tensor([1.0], dtype=mindspore.float32)
>>> sd_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>> ans = n1.log_prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = n1.log_prob(value, mean_b, sd_b)
>>> print(ans.shape)
(3,)
>>> # `mean` and `sd` must be passed in during function calls
>>> ans = n2.log_prob(value, mean_a, sd_a)
>>> print(ans.shape)
(3,)
```

log_prob(*value*, *mean=None*, *sd=None*)

Evaluate log probability of the value of the Laplace distribution.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor, optional) - the mean of the distribution. Default: `None` .
- **sd** (Tensor, optional) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

20.2.12 mindspore.nn.probability.distribution.Logistic

class mindspore.nn.probability.distribution.**Logistic**(*loc=None*, *scale=None*, *seed=None*, *dtype=mstype.float32*, *name='Logistic'*)

Logistic distribution. A Logistic distributio is a continuous distribution with the range $(-\inf, \inf)$ and the probability density function:

$$f(x, a, b) = 1/b \exp(\exp(-(x-a)/b) - x).$$

where a, b are loc and scale parameter respectively.

Parameters

- **loc** (*float*, *list*, *numpy.ndarray*, *Tensor*) –The location of the Logistic distribution. a in the formula. Default: `None` .
- **scale** (*float*, *list*, *numpy.ndarray*, *Tensor*) –The scale of the Logistic distribution. b in the formula. Default: `None` .
- **seed** (*int*) –The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (mindspore.dtype) –The type of the event samples. Default: `mstype.float32` .
- **name** (*str*) –The name of the distribution. Default: `'Logistic'` .

Note: *scale* must be greater than zero. *dist_spec_args* are *loc* and *scale*. *dtype* must be a float type because Logistic distributions are continuous.

Raises

- **ValueError** −When scale <= 0.
- **TypeError** −When the input *dtype* is not a float or a subclass of float.

Supported Platforms:

`Ascend` `GPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Logistic distribution of loc 3.0 and scale 4.0.
>>> l1 = msd.Logistic(3.0, 4.0, dtype=mindspore.float32)
>>> # A Logistic distribution can be initialized without arguments.
>>> # In this case, `loc` and `scale` must be passed in through arguments.
>>> l2 = msd.Logistic(dtype=mindspore.float32)
>>>
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> loc_a = Tensor([2.0], dtype=mindspore.float32)
>>> scale_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> loc_b = Tensor([1.0], dtype=mindspore.float32)
>>> scale_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>>
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`,
>>> # have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function
>>> ans = l1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to distribution b.
>>> ans = l1.prob(value, loc_b, scale_b)
>>> print(ans.shape)
(3,)
>>> # `loc` and `scale` must be passed in during function calls
>>> ans = l1.prob(value, loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `mode`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> # Example of `mean`. `mode`, `sd`, `var`, and `entropy` are similar.
>>> ans = l1.mean()
```

(continues on next page)

(continued from previous page)

```
>>> print(ans.shape)
()
>>> ans = l1.mean(loc_b, scale_b)
>>> print(ans.shape)
(3,)
>>> # `loc` and `scale` must be passed in during function calls.
>>> ans = l1.mean(loc_a, scale_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     loc (Tensor): the location of the distribution. Default: self.loc.
>>> #     scale (Tensor): the scale of the distribution. Default: self.scale.
>>> ans = l1.sample()
>>> print(ans.shape)
()
>>> ans = l1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = l1.sample((2,3), loc_b, scale_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = l1.sample((2,3), loc_a, scale_a)
>>> print(ans.shape)
(2, 3, 3)
```

property loc

Return the loc parameter of the distribution.

Returns

Tensor, the loc parameter of the distribution.

property scale

Return the scale parameter of the distribution.

Returns

Tensor, the scale parameter of the distribution.

cdf(*value*, *loc*, *scale*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy(*dist*, *loc_b*, *scale_b*, *loc*, *scale*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.

- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy(*loc*, *scale*)
Compute the value of the entropy.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss(*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf(*value*, *loc*, *scale*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *loc*, *scale*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *loc*, *scale*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*loc*, *scale*)

Compute the mean value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*loc*, *scale*)

Compute the mode value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *loc*, *scale*)

The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *loc*, *scale*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd (*loc*, *scale*)
The standard deviation.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function (*value*, *loc*, *scale*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var (*loc*, *scale*)
Compute the variance of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.13 mindspore.nn.probability.distribution.LogNormal

class mindspore.nn.probability.distribution.**LogNormal** (*loc=None*, *scale=None*, *seed=0*, *dtype=mstype.float32*, *name='LogNormal'*)

LogNormal distribution. A log-normal (or lognormal) distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. The log-normal distribution has the range $(0, \inf)$ with the pdf as

$$f(x, \mu, \sigma) = 1/x\sigma\sqrt{2\pi}\exp(-(\ln(x) - \mu)^2/2\sigma^2).$$

where μ, σ are the mean and the standard deviation of the underlying normal distribution respectively. It is constructed as the exponential transformation of a Normal distribution.

Parameters

- **loc** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The mean of the underlying Normal distribution. Default: `None` .
- **scale** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The standard deviation of the underlying Normal distribution. Default: `None` .

- **seed** (*int*) –the seed used in sampling. The global seed is used if it is None. Default: `0` .
- **dtype** (`mindspore.dtype`) –type of the distribution. Default: `mstype.float32` .
- **name** (*str*) –the name of the distribution. Default: `'LogNormal'` .

Note: *scale* must be greater than zero. *dist_spec_args* are *loc* and *scale*. *dtype* must be a float type because LogNormal distributions are continuous.

Raises

- **ValueError** –When scale <= 0.
- **TypeError** –When the input *dtype* is not a subclass of float.

Supported Platforms:

`Ascend GPU`

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> class Prob(nn.Cell):
...     def __init__(self):
...         super(Prob, self).__init__()
...         self.ln = msd.LogNormal(np.array([0.3]), np.array([[0.2], [0.4]]),␣
↪dtype=mindspore.float32)
...     def construct(self, x_):
...         return self.ln.prob(x_)
>>> pdf = Prob()
>>> output = pdf(Tensor([1.0, 2.0], dtype=mindspore.float32))
>>> print(output.shape)
(2, 2)
```

property loc

Return the loc parameter of the distribution.

Returns

Tensor, the loc parameter of the distribution.

property scale

Return the scale parameter of the distribution.

Returns

Tensor, the scale parameter of the distribution.

cdf (*value*, *loc*, *scale*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .

- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy (*loc*, *scale*)
Compute the value of the entropy.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss (*dist*, *loc_b*, *scale_b*, *loc*, *scale*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **loc_b** (Tensor) - the loc parameter of the other distribution.
- **scale_b** (Tensor) - the scale parameter of the other distribution.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf (*value*, *loc*, *scale*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *loc*, *scale*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *loc*, *scale*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*loc*, *scale*)
Compute the mean value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*loc*, *scale*)
Compute the mode value of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *loc*, *scale*)
The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *loc*, *scale*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*loc*, *scale*)
The standard deviation.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *loc*, *scale*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*loc*, *scale*)
Compute the variance of the distribution.

Parameters

- **loc** (Tensor) - the loc parameter of the distribution. Default: `None` .
- **scale** (Tensor) - the scale parameter of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.14 mindspore.nn.probability.distribution.Normal

class mindspore.nn.probability.distribution.**Normal**(*mean=None*, *sd=None*, *seed=None*, *dtype=mstype.float32*, *name='Normal'*)

Normal distribution. A Normal distribution is a continuous distribution with the range $(-\inf, \inf)$ and the probability density function:

$$f(x, \mu, \sigma) = 1/\sigma\sqrt{2\pi}\exp(-(x-\mu)^2/2\sigma^2).$$

where μ, σ are the mean and the standard deviation of the normal distribution respectively.

Parameters

- **mean** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The mean of the Normal distribution. Default: None .
- **sd** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The standard deviation of the Normal distribution. Default: None .
- **seed** (*int*) –The seed used in sampling. The global seed is used if it is None. Default: None .
- **dtype** (mindspore.dtype) –The type of the event samples. Default: mstype.float32 .
- **name** (*str*) –The name of the distribution. Default: 'Normal' .

Note: *sd* must be greater than zero. *dist_spec_args* are *mean* and *sd*. *dtype* must be a float type because Normal distributions are continuous.

Raises

- **ValueError** –When sd <= 0.
- **TypeError** –When the input *dtype* is not a subclass of float.

Supported Platforms:

Ascend GPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Normal distribution of the mean 3.0 and the standard deviation 4.0.
>>> n1 = msd.Normal(3.0, 4.0, dtype=mindspore.float32)
>>> # A Normal distribution can be initialized without arguments.
>>> # In this case, `mean` and `sd` must be passed in through arguments.
>>> n2 = msd.Normal(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> mean_a = Tensor([2.0], dtype=mindspore.float32)
>>> sd_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> mean_b = Tensor([1.0], dtype=mindspore.float32)
>>> sd_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
```

(continues on next page)

(continued from previous page)

```
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`,
>>> # have the same arguments as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     mean (Tensor): the mean of the distribution. Default: self._mean_value.
>>> #     sd (Tensor): the standard deviation of the distribution. Default: self._sd_value.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function
>>> ans = n1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = n1.prob(value, mean_b, sd_b)
>>> print(ans.shape)
(3,)
>>> # `mean` and `sd` must be passed in during function calls
>>> ans = n2.prob(value, mean_a, sd_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     mean (Tensor): the mean of the distribution. Default: self._mean_value.
>>> #     sd (Tensor): the standard deviation of the distribution. Default: self._sd_value.
>>> # Example of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = n1.mean() # return 0.0
>>> print(ans.shape)
()
>>> ans = n1.mean(mean_b, sd_b) # return mean_b
>>> print(ans.shape)
(3,)
>>> # `mean` and `sd` must be passed in during function calls.
>>> ans = n2.mean(mean_a, sd_a)
>>> print(ans.shape)
(3,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same:
>>> # Args:
>>> #     dist (str): the type of the distributions. Only "Normal" is supported.
>>> #     mean_b (Tensor): the mean of distribution b.
>>> #     sd_b (Tensor): the standard deviation of distribution b.
>>> #     mean_a (Tensor): the mean of distribution a. Default: self._mean_value.
>>> #     sd_a (Tensor): the standard deviation of distribution a. Default: self._sd_value.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = n1.kl_loss('Normal', mean_b, sd_b)
>>> print(ans.shape)
(3,)
>>> ans = n1.kl_loss('Normal', mean_b, sd_b, mean_a, sd_a)
>>> print(ans.shape)
(3,)
>>> # Additional `mean` and `sd` must be passed in.
>>> ans = n2.kl_loss('Normal', mean_b, sd_b, mean_a, sd_a)
>>> print(ans.shape)
(3,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     mean (Tensor): the mean of the distribution. Default: self._mean_value.
```

(continues on next page)

(continued from previous page)

```
>>> #     sd (Tensor): the standard deviation of the distribution. Default: self._sd_value.
>>> ans = n1.sample()
>>> print(ans.shape)
()
>>> ans = n1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = n1.sample((2,3), mean_b, sd_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = n2.sample((2,3), mean_a, sd_a)
>>> print(ans.shape)
(2, 3, 3)
```

property mean

Return the mean of the distribution.

Returns

Tensor, the mean of the distribution.

property sd

Return the standard deviation of the distribution.

Returns

Tensor, the standard deviation of the distribution.

cdf (*value*, *mean*, *sd*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *mean_b*, *sd_b*, *mean*, *sd*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **mean_b** (Tensor) - the mean of the other distribution.
- **sd_b** (Tensor) - the standard deviation of the other distribution.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy (*mean*, *sd*)

Compute the value of the entropy.

Parameters

- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss (*dist*, *mean_b*, *sd_b*, *mean*, *sd*)

Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **mean_b** (Tensor) - the mean of the other distribution.
- **sd_b** (Tensor) - the standard deviation of the other distribution.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf (*value*, *mean*, *sd*)

Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob (*value*, *mean*, *sd*)

the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival (*value*, *mean*, *sd*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mode(*mean*, *sd*)

Compute the mode value of the distribution.

Parameters

- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *mean*, *sd*)

The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *mean*, *sd*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the sample.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

survival_function(*value*, *mean*, *sd*)

Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*mean*, *sd*)

Compute the variance of the distribution.

Parameters

- **mean** (Tensor) - the mean of the distribution. Default: `None` .
- **sd** (Tensor) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.15 mindspore.nn.probability.distribution.Poisson

class mindspore.nn.probability.distribution.**Poisson**(*rate=None*, *seed=None*, *dtype=mstype.float32*, *name='Poisson'*)

Poisson Distribution. A Poisson Distribution is a discrete distribution with the range as the non-negative integers, and the probability mass function as

$$P(X = k) = \lambda^k \exp(-\lambda)/k!, k = 1, 2, ...$$

where λ is the rate of the distribution.

Parameters

- **rate** (*list, numpy.ndarray, Tensor*) –The rate of the Poisson distribution. λ in the formula. Default: `None` .
- **seed** (*int*) –The seed used in sampling. The global seed is used if it is `None` . Default: `None` .
- **dtype** (`mindspore.dtype`) –The type of the event samples. Default: `mstype.float32` .
- **name** (*str*) –The name of the distribution. Default: `'Poisson'` .

Note: *rate* must be strictly greater than 0. *dist_spec_args* is *rate*.

Raises

ValueError –When rate <= 0.

Supported Platforms:

`Ascend`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize an Poisson distribution of the rate 0.5.
>>> p1 = msd.Poisson([0.5], dtype=mindspore.float32)
>>> # An Poisson distribution can be initialized without arguments.
>>> # In this case, `rate` must be passed in through `args` during function calls.
>>> p2 = msd.Poisson(dtype=mindspore.float32)
>>>
>>> # Here are some tensors used below for testing
>>> value = Tensor([1, 2, 3], dtype=mindspore.int32)
>>> rate_a = Tensor([0.6], dtype=mindspore.float32)
>>> rate_b = Tensor([0.2, 0.5, 0.4], dtype=mindspore.float32)
>>>
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
```

(continues on next page)

(continued from previous page)

```
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`, are the
↪same as follows.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     rate (Tensor): the rate of the distribution. Default: self.rate.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing `prob` by the name of the function.
>>> ans = p1.prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to distribution b.
>>> ans = p1.prob(value, rate_b)
>>> print(ans.shape)
(3,)
>>> # `rate` must be passed in during function calls.
>>> ans = p2.prob(value, rate_a)
>>> print(ans.shape)
(3,)
>>> # Functions `mean`, `mode`, `sd`, and 'var' have the same arguments as follows.
>>> # Args:
>>> #     rate (Tensor): the rate of the distribution. Default: self.rate.
>>> # Examples of `mean`, `sd`, `mode`, and `var` are similar.
>>> ans = p1.mean() # return 2
>>> print(ans.shape)
(1,)
>>> ans = p1.mean(rate_b) # return 1 / rate_b
>>> print(ans.shape)
(3,)
>>> # `rate` must be passed in during function calls.
>>> ans = p2.mean(rate_a)
>>> print(ans.shape)
(1,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     probs1 (Tensor): the rate of the distribution. Default: self.rate.
>>> ans = p1.sample()
>>> print(ans.shape)
(1, )
>>> ans = p1.sample((2,3))
>>> print(ans.shape)
(2, 3, 1)
>>> ans = p1.sample((2,3), rate_b)
>>> print(ans.shape)
(2, 3, 3)
>>> ans = p2.sample((2,3), rate_a)
>>> print(ans.shape)
(2, 3, 1)
```

property rate

Return rate parameter.

Returns

Tensor, the value of the rate.

cdf(*value*, *rate*)
Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

log_cdf(*value*, *rate*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*, *rate*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival(*value*, *rate*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*rate*)
Compute the mean value of the distribution.

Parameters

- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*rate*)
Compute the mode value of the distribution.

Parameters

- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *rate*)

The probability of the given value. For the discrete distribution, it is the probability mass function(pmf).

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *rate*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the tensor.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*rate*)

The standard deviation.

Parameters

- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *rate*)

Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*rate*)

Compute the variance of the distribution.

Parameters

- **rate** (Tensor) - the value of the rate. Default: `None` .

Returns

Tensor, the variance of the distribution.

20.2.16 mindspore.nn.probability.distribution.StudentT

class mindspore.nn.probability.distribution.**StudentT**(*df=None*, *mean=None*, *sd=None*, *seed=None*, *dtype=mstype.float32*, *name='StudentT'*)

StudentT distribution. A StudentT distribution is a continuous distribution with the range $(-\inf, \inf)$ and the probability density function:

$$f(x, \nu, \mu, \sigma) = (1 + y^2/\nu)^{(-0.5*(\nu+1))}/Z$$

where $y = (x - \mu)/\sigma$, $Z = abs(\sigma) * \sqrt{(\nu * \pi)} * \Gamma(0.5 * \nu)/\Gamma(0.5 * (\nu + 1))$, ν, μ, σ are the degrees of freedom , mean and sd of the laplace distribution respectively.

Parameters

- **df** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) –The degrees of freedom. If this arg is `None` , then the df of the distribution will be passed in runtime. Default: `None` .
- **mean** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) –The mean of the distribution. If this arg is `None` , then the df of the distribution will be passed in runtime. Default: `None` .
- **sd** (*Union[int, float, list, numpy.ndarray, Tensor], optional*) –The standard deviation of the distribution. If this arg is `None` , then the sd of the distribution will be passed in runtime. Default: `None` .
- **seed** (*int, optional*) –The seed used in sampling. The global seed is used if it is None. Default: `None` .
- **dtype** (*mindspore.dtype, optional*) –The type of the event samples. Default: `mstype.float32` .
- **name** (*str, optional*) –The name of the distribution. Default: `'StudentT'` .

Note:

- *df* must be greater than zero.
- *sd* must be greater than zero.
- *dtype* must be a float type because StudentT distributions are continuous.
- If the arg *df*, *mean* or *sd* is passed in runtime, then it will be used as the parameter value. Otherwise, the value passed in the constructor will be used.

Raises

- **ValueError** –When df <= 0.
- **ValueError** –When sd <= 0.
- **TypeError** –When the input *dtype* is not a subclass of float.

Supported Platforms:

`CPU`

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a StudentT distribution of the df 2.0, the mean 3.0 and the standard␣
↪deviation 4.0.
>>> n1 = msd.StudentT(2.0, 3.0, 4.0, dtype=mindspore.float32)
>>> # A StudentT distribution can be initialized without arguments.
>>> # In this case, `df`, `mean` and `sd` must be passed in through arguments.
>>> n2 = msd.StudentT(dtype=mindspore.float32)
>>> # Here are some tensors used below for testing
>>> value = Tensor([1.0, 2.0, 3.0], dtype=mindspore.float32)
>>> df_a = Tensor([2.0], dtype=mindspore.float32)
>>> mean_a = Tensor([2.0], dtype=mindspore.float32)
>>> sd_a = Tensor([2.0, 2.0, 2.0], dtype=mindspore.float32)
>>> df_b = Tensor([1.0], dtype=mindspore.float32)
>>> mean_b = Tensor([1.0], dtype=mindspore.float32)
>>> sd_b = Tensor([1.0, 1.5, 2.0], dtype=mindspore.float32)
>>> ans = n1.log_prob(value)
>>> print(ans.shape)
(3,)
>>> # Evaluate with respect to the distribution b.
>>> ans = n1.log_prob(value, df_b, mean_b, sd_b)
>>> print(ans.shape)
(3,)
>>> # `mean` and `sd` must be passed in during function calls
>>> ans = n2.log_prob(value, df_a, mean_a, sd_a)
>>> print(ans.shape)
(3,)
```

log_prob(*value*, *df=None*, *mean=None*, *sd=None*)

Evaluate log probability of the value of the StudentT distribution.

Parameters

- **value** (Tensor) - the value to compute.
- **df** (Tensor, optional) - the degrees of freedom of the distribution. Default: `None` .
- **mean** (Tensor, optional) - the mean of the distribution. Default: `None` .
- **sd** (Tensor, optional) - the standard deviation of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

20.2.17 mindspore.nn.probability.distribution.TransformedDistribution

class mindspore.nn.probability.distribution.**TransformedDistribution**(*bijector*, *distribution*, *seed=None*, *name='transformed_distribution'*)

Transformed Distribution. This class contains a bijector and a distribution and transforms the original distribution to a new distribution through the operation defined by the bijector. If X is an random variable following the underying distribution, and $g(x)$ is a function represented by the bijector, then $Y = g(X)$ is a random variable following the transformed distribution.

Parameters

- **bijector** (Bijector) –The transformation to perform.
- **distribution** (Distribution) –The original distribution. Must be a float dtype.
- **seed** (*int*) –The seed is used in sampling. The global seed is used if it is None. Default: None . If this seed is given when a TransformedDistribution object is initialized, the object's sampling function will use this seed; elsewise, the underlying distribution's seed will be used.
- **name** (*str*) –The name of the transformed distribution. Default: 'transformed_distribution' .

Note: The arguments used to initialize the original distribution cannot be None. For example, mynormal = msd.Normal(dtype=mindspore.float32) cannot be used to initialized a TransformedDistribution since *mean* and *sd* are not specified. *batch_shape* is the batch_shape of the original distribution. *broadcast_shape* is the broadcast shape between the original distribution and bijector. *is_scalar_batch* is only true if both the original distribution and the bijector are scalar batches. *default_parameters*, *parameter_names* and *parameter_type* are set to be consistent with the original distribution. Derived class can overwrite *default_parameters* and *parameter_names* by calling *reset_parameters* followed by *add_parameter*.

Raises

- **TypeError** –When the input *bijector* is not a Bijector instance.
- **TypeError** –When the input *distribution* is not a Distribution instance.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import numpy as np
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> import mindspore.nn.probability.bijector as msb
>>> from mindspore import Tensor
>>> class Net(nn.Cell):
...     def __init__(self, shape, dtype=mindspore.float32, seed=0, name='transformed_
↪distribution'):
...         super(Net, self).__init__()
...         # create TransformedDistribution distribution
...         self.exp = msb.Exp()
...         self.normal = msd.Normal(0.0, 1.0, dtype=dtype)
...         self.lognormal = msd.TransformedDistribution(self.exp, self.normal, seed=seed,␣
↪name=name)
...         self.shape = shape
```

(continues on next page)

(continued from previous page)

```
...
...     def construct(self, value):
...         cdf = self.lognormal.cdf(value)
...         sample = self.lognormal.sample(self.shape)
...         return cdf, sample
>>> shape = (2, 3)
>>> net = Net(shape=shape, name="LogNormal")
>>> x = np.array([2.0, 3.0, 4.0, 5.0]).astype(np.float32)
>>> tx = Tensor(x, dtype=mindspore.float32)
>>> cdf, sample = net(tx)
>>> print(sample.shape)
(2, 3)
```

property bijector
Return the bijector.

Returns

Bijector, the bijector.

property distribution
Return the distribution before transformation.

Returns

Distribution, the distribution before transformation.

property dtype
Return the data type of distribution.

Returns

mindspore.dtype, the data type of distribution.

property is_linear_transformation
Return whether the bijector is linear.

Returns

Bool, return True if the bijector is linear, otherwise return False.

cdf(*value*)
Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

log_cdf(*value*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob(*value*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the log value of the probability.

log_survival(*value*)

Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value of the K-L loss.

mean()

Compute the mean value of the distribution.

Returns

Tensor, the mean of the distribution.

prob(*value*)

The probability of the given value.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value of the probability.

sample(*shape*)

Generate samples.

Parameters

- **shape** (tuple) - the shape of the tensor.

Returns

Tensor, the sample following the distribution.

survival_function(*value*)

Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.

Returns

Tensor, the value of the survival function.

20.2.18 mindspore.nn.probability.distribution.Uniform

class mindspore.nn.probability.distribution.**Uniform**(*low=None*, *high=None*, *seed=None*, *dtype=mstype.float32*, *name='Uniform'*)

Uniform Distribution. A Uniform distributio is a continuous distribution with the range $[a, b]$ and the probability density function:

$$f(x, a, b) = 1/(b - a)$$

Where a, b are the lower and upper bound respectively.

Parameters

- **low** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The lower bound of the distribution. a in the formula. Default: None .
- **high** (*int*, *float*, *list*, *numpy.ndarray*, Tensor) –The upper bound of the distribution. b in the formula. Default: None .
- **seed** (*int*) –The seed uses in sampling. The global seed is used if it is None . Default: None .
- **dtype** (mindspore.dtype) –The type of the event samples. Default: mstype.float32 .
- **name** (*str*) –The name of the distribution. Default: 'Uniform' .

Note: *low* must be strictly less than *high*. *dist_spec_args* are *high* and *low*. *dtype* must be float type because Uniform distributions are continuous.

Raises

- **ValueError** –When high <= low.
- **TypeError** –When the input *dtype* is not a float or a subclass of float.

Supported Platforms:

Ascend GPU CPU

Examples

```
>>> import mindspore
>>> import mindspore.nn as nn
>>> import mindspore.nn.probability.distribution as msd
>>> from mindspore import Tensor
>>> # To initialize a Uniform distribution of the lower bound 0.0 and the higher bound 1.0.
>>> u1 = msd.Uniform(0.0, 1.0, dtype=mindspore.float32)
>>> # A Uniform distribution can be initialized without arguments.
>>> # In this case, `high` and `low` must be passed in through arguments during function␣
↪calls.
>>> u2 = msd.Uniform(dtype=mindspore.float32)
>>>
>>> # Here are some tensors used below for testing
>>> value = Tensor([0.5, 0.8], dtype=mindspore.float32)
>>> low_a = Tensor([0., 0.], dtype=mindspore.float32)
>>> high_a = Tensor([2.0, 4.0], dtype=mindspore.float32)
>>> low_b = Tensor([-1.5], dtype=mindspore.float32)
>>> high_b = Tensor([2.5, 5.], dtype=mindspore.float32)
>>> # Private interfaces of probability functions corresponding to public interfaces,␣
↪including
```

(continues on next page)

(continued from previous page)

```
>>> # `prob`, `log_prob`, `cdf`, `log_cdf`, `survival_function`, and `log_survival`, have␣
↪the same arguments.
>>> # Args:
>>> #     value (Tensor): the value to be evaluated.
>>> #     low (Tensor): the lower bound of the distribution. Default: self.low.
>>> #     high (Tensor): the higher bound of the distribution. Default: self.high.
>>> # Examples of `prob`.
>>> # Similar calls can be made to other probability functions
>>> # by replacing 'prob' by the name of the function.
>>> ans = u1.prob(value)
>>> print(ans.shape)
(2,)
>>> # Evaluate with respect to distribution b.
>>> ans = u1.prob(value, low_b, high_b)
>>> print(ans.shape)
(2,)
>>> # `high` and `low` must be passed in during function calls.
>>> ans = u2.prob(value, low_a, high_a)
>>> print(ans.shape)
(2,)
>>> # Functions `mean`, `sd`, `var`, and `entropy` have the same arguments.
>>> # Args:
>>> #     low (Tensor): the lower bound of the distribution. Default: self.low.
>>> #     high (Tensor): the higher bound of the distribution. Default: self.high.
>>> # Examples of `mean`. `sd`, `var`, and `entropy` are similar.
>>> ans = u1.mean() # return 0.5
>>> print(ans.shape)
()
>>> ans = u1.mean(low_b, high_b) # return (low_b + high_b) / 2
>>> print(ans.shape)
(2,)
>>> # `high` and `low` must be passed in during function calls.
>>> ans = u2.mean(low_a, high_a)
>>> print(ans.shape)
(2,)
>>> # Interfaces of 'kl_loss' and 'cross_entropy' are the same.
>>> # Args:
>>> #     dist (str): the type of the distributions. Should be "Uniform" in this case.
>>> #     low_b (Tensor): the lower bound of distribution b.
>>> #     high_b (Tensor): the upper bound of distribution b.
>>> #     low_a (Tensor): the lower bound of distribution a. Default: self.low.
>>> #     high_a (Tensor): the upper bound of distribution a. Default: self.high.
>>> # Examples of `kl_loss`. `cross_entropy` is similar.
>>> ans = u1.kl_loss('Uniform', low_b, high_b)
>>> print(ans.shape)
(2,)
>>> ans = u1.kl_loss('Uniform', low_b, high_b, low_a, high_a)
>>> print(ans.shape)
(2,)
>>> # Additional `high` and `low` must be passed in.
>>> ans = u2.kl_loss('Uniform', low_b, high_b, low_a, high_a)
>>> print(ans.shape)
(2,)
>>> # Examples of `sample`.
>>> # Args:
>>> #     shape (tuple): the shape of the sample. Default: ()
>>> #     low (Tensor): the lower bound of the distribution. Default: self.low.
```

(continues on next page)

(continued from previous page)

```
>>> #     high (Tensor): the upper bound of the distribution. Default: self.high.
>>> ans = u1.sample()
>>> print(ans.shape)
()
>>> ans = u1.sample((2,3))
>>> print(ans.shape)
(2, 3)
>>> ans = u1.sample((2,3), low_b, high_b)
>>> print(ans.shape)
(2, 3, 2)
>>> ans = u2.sample((2,3), low_a, high_a)
>>> print(ans.shape)
(2, 3, 2)
```

property high

Return the upper bound of the distribution.

Returns

Tensor, the upper bound of the distribution.

property low

Return the upper bound of the distribution.

Returns

Tensor, the lower bound of the distribution.

cdf (*value*, *high*, *low*)

Compute the cumulatuve distribution function(CDF) of the given value.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the cumulatuve distribution function for the given input.

cross_entropy (*dist*, *high_b*, *low_b*, *high*, *low*)

Compute the cross entropy of two distribution.

Parameters

- **dist** (str) - the type of the other distribution.
- **high_b** (Tensor) - the upper bound of the other distribution.
- **low_b** (Tensor) - the lower bound of the other distribution.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the cross entropy.

entropy (*high*, *low*)

Compute the value of the entropy.

Parameters

- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the entropy.

kl_loss (*dist*, *high_b*, *low_b*, *high*, *low*)
Compute the value of the K-L loss between two distribution, namely KL(a||b).

Parameters

- **dist** (str) - the type of the other distribution.
- **high_b** (Tensor) - the upper bound of the other distribution.
- **low_b** (Tensor) - the lower bound of the other distribution.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

log_cdf (*value*, *high*, *low*)
Compute the log value of the cumulatuve distribution function.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the log value of the cumulatuve distribution function.

log_prob (*value*, *high*, *low*)
the log value of the probability.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the log value of the probability.

log_survival (*value*, *high*, *low*)
Compute the log value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the K-L loss.

mean(*high*, *low*)
Compute the mean value of the distribution.

Parameters

- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the mean of the distribution.

mode(*high*, *low*)
Compute the mode value of the distribution.

Parameters

- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the mode of the distribution.

prob(*value*, *high*, *low*)
The probability of the given value. For the continuous distribution, it is the probability density function.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the probability.

sample(*shape*, *high*, *low*)
Generate samples.

Parameters

- **shape** (tuple) - the shape of the tensor.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the sample following the distribution.

sd(*high*, *low*)
The standard deviation.

Parameters

- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the standard deviation of the distribution.

survival_function(*value*, *high*, *low*)
Compute the value of the survival function.

Parameters

- **value** (Tensor) - the value to compute.
- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the value of the survival function.

var(*high*, *low*)
Compute the variance of the distribution.

Parameters

- **high** (Tensor) - the upper bound of the distribution. Default: `None` .
- **low** (Tensor) - the lower bound of the distribution. Default: `None` .

Returns

Tensor, the variance of the distribution.

CHAPTER

TWENTYONE

MINDSPORE.REWRITE

MindSpore ReWrite module allows users to modify the network's forward computation process by customizing rules to support operations such as inserting, deleting and replacing statements.

For a quick start of using ReWrite, please refer to Modifying Network With ReWrite .

MindSpore Rewrite package. This is an experimental python package that is subject to change or deletion.

class mindspore.rewrite.**Node**(*node: NodeImpl*)

A node (Node) can be understood as a basic data structure unit in the computational graph of a neural network, which represents an operation or computational step in the network. Each node usually corresponds to a statement or expression in the source code, which contains the information needed to perform the operation, such as the type of operation, input data, output result, and connection relationships with other nodes.

Nodes can express a `Cell` call statement, a `Primitive` call statement, an arithmetic operation statement, a return statements, etc. of the forward calculation process.

Parameters

node (*NodeImpl*) – A handler of *NodeImpl*. It is recommended to call the specific methods in Node to create a Node, such as 'create_call_cell', rather than calling the Node's constructor directly. Don't care what *NodeImpl* is, just treat it as a handle.

static create_call_cell(*cell:* Cell, *targets: List[Union[*ScopedValue*, str]]*, *args: List[*ScopedValue*] = None*, *kwargs: Dict[str,* ScopedValue*] = None*, *name: str = ''*, *is_sub_net: bool = False*)

Create a node. Only support create from a *Cell* now.

A node is corresponding to source code like:

```
targets = self.name(*args, **kwargs)
```

Parameters

- **cell** (`Cell`) – Cell-operator of this forward-layer.
- **targets** (*List[Union[*`ScopedValue`*, str]]*) – Indicate output names. Used as targets of an assign statement in source code.
- **args** (*List[*`ScopedValue`*]*) – Indicate input names. Used as args of a call expression of an assign statement in source code. Default: `None` , which indicates the *cell* has no args inputs.
- **kwargs** (*Dict[str,* `ScopedValue`*]*) – Used as kwargs of a call expression of an assign statement in source code. Indicate keyword input names. Type of key must be *str* and type of value must be *ScopedValue*. Default: `None` , which indicates the *cell* has no kwargs inputs.
- **name** (*str*) – Indicate the name of node. Used as field name in source code. Default is None. Rewrite will generate name from *cell* when name is None. Rewrite will check and ensure the uniqueness of *name* while node being inserted. Default: `""` .
- **is_sub_net** (*bool*) – Indicate that is *cell* a network. If *is_sub_net* is true, Rewrite will try to parse the *cell* to a TreeNode, otherwise the *cell* is parsed to a CallCell node. Default: `False` .

Returns

An instance of *Node*.

Raises

- **TypeError** – If *cell* is not a *Cell*.
- **TypeError** – If *targets* is not *list*.
- **TypeError** – If the type of *targets* is not in *[ScopedValue, str]*.
- **TypeError** – If arg in *args* is not a *ScopedValue*.
- **TypeError** – If key of *kwargs* is not a str or value of kwarg in *kwargs* is not a *ScopedValue*.

Examples

```
>>> from mindspore.rewrite import SymbolTree, ScopedValue
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> position = stree.after(node)
>>> new_node = node.create_call_cell(cell=nn.ReLU(), targets=['x'],
...                                  args=[ScopedValue.create_naming_value('x')], name=
↪'new_relu')
>>> stree.insert(position, new_node)
>>> print(type(new_node))
<class 'mindspore.rewrite.api.node.Node'>
```

static create_call_function(*function: FunctionType*, *targets: List[Union[*ScopedValue*, str]]*, *args: List[*ScopedValue*] = None*, *kwargs: Dict[str,* ScopedValue*] = None*)

Create a node that corresponds to a function call.

Note: The codes inside the function will not be parsed.

Parameters

- **function** (*FunctionType*) – The function to be called.
- **targets** (*List[Union[ScopedValue, str]]*) – indicates output names. Used as targets of an assign statement in source code.
- **args** (*List[ScopedValue]*) – Indicate input names. Used as args of a call expression of an assign statement in source code. Default: `None` , which indicates the *function* has no args inputs.
- **kwargs** (*Dict[str, ScopedValue]*) – Used as kwargs of a call expression of an assign statement in source code. Indicate keyword input names. Type of key must be *str* and type of value must be *ScopedValue*. Default: `None` , which indicates the *function* has no kwargs inputs.

Returns

An instance of *Node*.

Raises

- **TypeError** – If *function* is not a *FunctionType*.

- **TypeError** – If *targets* is not *list*.
- **TypeError** – If the type of *targets* is not in *[ScopedValue, str]*.
- **TypeError** – If arg in *args* is not a *ScopedValue*.
- **TypeError** – If key of *kwargs* is not a str or value of kwarg in *kwargs* is not a *ScopedValue*.

Examples

```
>>> from mindspore.rewrite import SymbolTree, ScopedValue
>>> import mindspore.nn as nn
>>> from mindspore import ops
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> position = stree.after(node)
>>> new_node = node.create_call_function(function=ops.abs, targets=['x'],
...                                      args=[ScopedValue.create_naming_value('x')])
>>> stree.insert(position, new_node)
>>> print(new_node.get_node_type())
NodeType.CallFunction
```

get_args()

Get arguments of current node.

Returns

A list of arguments of type `ScopedValue` .

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> print(node.get_args())
[x]
```

get_inputs()

Gets a list of nodes whose output values are used as input values for the current node.

Returns

A list of nodes.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv2")
>>> inputs = node.get_inputs()
>>> print([input.get_name() for input in inputs])
['max_pool2d']
```

get_instance_type()

Gets the instance type called in the code corresponding to the current node.

- When *node_type* of current node is *CallCell*, the code for that node calls an instance of type `Cell` .
- When *node_type* of current node is *CallPrimitive*, the code for that node calls an instance of type `Primitive` .
- When *node_type* of current node is *Tree*, the code for that node calls an instance of network type.
- When *node_type* of current node is *Python*, *Input*, *Output* or *CallMethod*, the instance type is `NoneType` .

Returns

The type of instance called in the statement corresponding to the current node.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> instance_type = node.get_instance_type()
>>> print(instance_type)
<class 'mindspore.nn.layer.conv.Conv2d'>
```

get_kwargs()

Get keyword arguments of current node.

Returns

A dict of keyword arguments, where key is of type str, and value is of type `ScopedValue` .

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> from mindspore import nn
>>>
>>> class ReLUNet(nn.Cell):
...     def __init__(self):
...         super().__init__()
...         self.relu = nn.ReLU()
...
```

(continues on next page)

(continued from previous page)

```
...     def construct(self, input):
...         output = self.relu(input=input)
...         return output
>>>
>>> net = ReLUNet()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("relu")
>>> print(node.get_kwargs())
{'input': input}
```

get_name()

Get the name of current node.

When node has been inserted into *SymbolTree*, the name of node should be unique in *SymbolTree*.

Returns

A string as name of node.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> name = node.get_name()
>>> print(name)
conv1
```

get_node_type()

Get the node_type of current node. See *mindspore.rewrite.NodeType* for details on node types.

Returns

A NodeType as node_type of node.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> node_type = node.get_node_type()
>>> print(node_type)
NodeType.CallCell
```

get_sub_tree()

Get the sub symbol tree stored in node with type of *NodeType.Tree* . See *mindspore.rewrite.NodeType* for details on node types.

Returns

SymbolTree stored in Tree node.

Raises

- **TypeError** – If current node is not type of *NodeType.Tree* .
- **AttributeError** – If no symbol tree is stored in Tree node.

Examples: >>> import mindspore.nn as nn >>> from mindspore.rewrite import SymbolTree >>> >>> class SubNet(nn.Cell): ···def __init__(self): ···super().__init__() ···self.relu = nn.ReLU() ·····def construct(self, x): ···x = self.relu(x) ···return x ···>>> class Net(nn.Cell): ···def __init__(self): ···super().__init__() ···self.subnet = SubNet() ····· def construct(self, x): ···x = self.subnet(x) ···return x >>> >>> net = Net() >>> stree = SymbolTree.create(net) >>> node = stree.get_node("subnet") >>> print(type(node.get_sub_tree())) <class 'mindspore.rewrite.api.symbol_tree.SymbolTree'>

get_symbol_tree()

Get the symbol tree which current node belongs to.

Returns

SymbolTree, None if current node does not belong to any SymbolTree.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> print(type(node.get_symbol_tree()))
<class 'mindspore.rewrite.api.symbol_tree.SymbolTree'>
```

get_targets()

Gets a list of output values for the current node.

Returns

A list of outputs of type `ScopedValue` .

get_users()

Get a list of nodes that use the output of the current node as input.

Returns

A list of nodes.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> users = node.get_users()
>>> print([user.get_name() for user in users])
['relu']
```

set_arg(*index: int*, *arg: Union[ScopedValue, str]*)
Set argument of current node.

Parameters

- **index** (*int*) –Indicate which input being modified.
- **arg** (*Union[ScopedValue, str]*) –New argument to been set.

Raises

- **TypeError** –If *index* is not a *int* number.
- **TypeError** –If the type of *arg* is not in [*ScopedValue*, *str*].

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("relu_3")
>>> node.set_arg(0, "fc1")
>>> print(node.get_args())
[fc1]
```

set_arg_by_node(*arg_idx: int*, *src_node: 'Node'*, *out_idx: Optional[int] = None*)
Set argument of current node by another Node.

Parameters

- **arg_idx** (*int*) –Indicate which input being modified.
- **src_node** (Node) –A *Node* as new input. Can be a node or name of node.
- **out_idx** (*int, optional*) –Indicate which output of *src_node* as new input of current node. Default: None , which means use first output of *src_node* as new input.

Raises

- **TypeError** –If *arg_idx* is not a *int* number.
- **ValueError** –If *arg_idx* is out of range.
- **TypeError** –If *src_node* is not a *Node* instance.
- **TypeError** –If *out_idx* is not a *int* number.
- **ValueError** –If *out_idx* is out of range.
- **ValueError** –If *src_node* has multi-outputs while *out_idx* is None or *out_idx* is not offered.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> src_node = stree.get_node("fc1")
>>> dst_node = stree.get_node("relu_3")
>>> dst_node.set_arg_by_node(0, src_node, 0)
>>> print(dst_node.get_args())
[fc1_var]
```

class mindspore.rewrite.**NodeType**

NodeType represents type of *Node*.

- Unknown: Not inited NodeType.
- CallCell: *CallCell* node represents invoking cell-op in forward method.
- CallPrimitive: *CallPrimitive* node represents invoking primitive-op in forward method.
- CallFunction: *CallFunction* node represents invoking a function in forward method.
- CallMethod: *CallMethod* node represents invoking of method in forward method which can not be mapped to cell-op or primitive-op in MindSpore.
- Python: *Python* node holds unsupported-ast-node or unnecessary-to-parse-ast-node.
- Input: *Input* node represents input of *SymbolTree* corresponding to arguments of forward method.
- Output: *Output* node represents output of SymbolTree corresponding to return statement of forward method.
- Tree: *Tree* node represents sub-network invoking in forward method.
- CellContainer: *CellContainer* node represents invoking method *mindspore.nn.SequentialCell* in forward method.
- MathOps: *MathOps* node represents a mathematical operation, such as adding or comparing in forward method.
- ControlFlow: *ControlFlow* node represents a control flow statement, such as if statement.

class mindspore.rewrite.**ScopedValue**(*arg_type:* ValueType, *scope: str = ''*, *value=None*)

ScopedValue represents a value with its full-scope.

ScopedValue is used to express: a left-value such as target of an assign statement, or a callable object such as func of a call statement, or a right-value such as args and kwargs of an assign statement.

Parameters

- **arg_type** (ValueType) –A *ValueType* represents type of current value.
- **scope** (*str, optional*) –A string represents scope of current value. Take "self.var1" as an example, *scope* of this var1 is "self". Default: "" .
- **value** –A handler represents value of current value. The type of value is corresponding to *arg_type*. Default: None .

static create_name_values(*names: Union[List[str], Tuple[str]]*, *scopes: Union[List[str], Tuple[str]] = None*)

Create a list of naming *ScopedValue*.

Parameters

- **names** (*List[str] or Tuple[str]*) –List or tuple of *str* represents names of referenced variables.

- **scopes** (*List[str] or Tuple[str], optional*) – List or tuple of *str* represents scopes of referenced variables. Default: `None` .

Returns

An list of instance of *ScopedValue*.

Raises

- **TypeError** – If *names* is not *list* or *tuple* and name in *names* is not *str*.
- **TypeError** – If *scopes* is not *list* or *tuple* and scope in *scopes* is not *str*.
- **ValueError** – If the length of names is not equal to the length of scopes when scopes are not None.

Examples

```
>>> from mindspore.rewrite import ScopedValue
>>> variables = ScopedValue.create_name_values(names=["z", "z_1"], scopes=["self", "self
↪"])
>>> print(variables)
[self.z, self.z_1]
```

classmethod create_naming_value(*name: str*, *scope: str = ''*)

Create a naming *ScopedValue*. A *NamingValue* represents a reference to another variable.

Parameters

- **name** (*str*) – A string represents the identifier of another variable.
- **scope** (*str, optional*) – A string represents the scope of another variable. Default: `""` .

Returns

An instance of *ScopedValue*.

Raises

- **TypeError** – If *name* is not *str*.
- **TypeError** – If *scope* is not *str*.

Examples

```
>>> from mindspore.rewrite import ScopedValue
>>> variable = ScopedValue.create_naming_value("conv", "self")
>>> print(variable)
self.conv
```

classmethod create_variable_value(*value*)

Create *ScopedValue* from a variable.

ScopedValue's type is determined by type of value. *ScopedValue*'s scope is empty.

Parameters

value – The value to be converted to *ScopedValue*.

Returns

An instance of *ScopedValue*.

Examples

```
>>> from mindspore.rewrite import ScopedValue
>>> variable = ScopedValue.create_variable_value(2)
>>> print(variable)
2
```

class mindspore.rewrite.**SymbolTree**(*handler: SymbolTreeImpl*)

SymbolTree stores information about a network, including statements of the network's forward computation process and the topological relationship between statement input and output.

The statements in the network are saved in the SymbolTree in the form of nodes, and by processing the nodes in the SymbolTree, you can delete the network code, insert and replace it, and get the modified network code and network instances.

Parameters

handler (*SymbolTreeImpl*) – SymbolTree internal implementation instance. It is recommended to call the *create* method in SymbolTree to create a SymbolTree, rather than calling SymbolTree's constructor directly. Don't care what *SymbolTreeImpl* is, just treat it as a handle.

after(*node: Union[Node, str]*)

Returns a location information after *node*. The return value of this interface is used as a parameter for the insert operation.

Parameters

node (*Union[Node, str]*) – Indicate the position after which node. Can be a node or name of node.

Returns

A *Position* to indicate where to insert node.

Raises

TypeError – If *node* is not a Node or str.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> for node in stree.nodes():
...     if node.get_name() == "conv1":
...         position = stree.after(node)
```

before(*node: Union[Node, str]*)

Returns a location information before *node*. The return value of this interface is used as a parameter for the insert operation.

Parameters

node (*Union[Node, str]*) – Indicate the position before which node. Can be a node or name of node.

Returns

A *Position* to indicate where to insert node.

Raises

TypeError – if *node* is not a Node or str.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> for node in stree.nodes():
...     if node.get_name() == "conv1":
...         position = stree.before(node)
```

classmethod create(*network*)

Create a SymbolTree object by passing in the network instance *network*.

This interface parses the *network* instance, expands each source code statement of the forward computation process, and parses it into nodes, which is stored in the SymbolTree. The specific process is as follows:

1. Obtain the source code of the network instance.
2. Perform AST parsing on the network and obtain the AST nodes (abstract syntax trees) of each statement in the network.
3. Expand complex statements in the network forward evaluation process into multiple simple statements.
4. Create a SymbolTree object. Each SymbolTree corresponds to one network instance.
5. Use the rewrite node to store each statement of the network forward computation process. The node records the input, output, and other information of the statement.
6. Save the rewrite node to the SymbolTree, and update and maintain the topological connection between the nodes.
7. Return the SymbolTree object corresponding to the network instance.

If a user-defined network of type *mindspore.nn.Cell* is called in the forward computation process of the network, rewrite will generate a node of type *NodeType.Tree* for the corresponding statement. This type of node stores a new SymbolTree, which parses and maintains the node information of the user-defined network.

If the following types of statements are called in the forward computation process of the network, rewrite will parse the internal statements in the statement and generate corresponding nodes:

- *mindspore.nn.SequentialCell*
- Functions(Excludes Python built-in functions and third-party library functions)
- Control flow statements(such as *if* statements)

Note: Because the specific execution branch of control flows are still unknown during the rewrite operation of the network, no topology information will be established between the nodes inside the control flow and the nodes outside. Users cannot obtain nodes inside the control flow when they acquire nodes outside the control flow using interfaces like *mindspore.rewrite.Node.get_inputs()* and *mindspore.rewrite.Node.get_users()* . Users also cannot obtain nodes outside the control flow, if they use these interfaces inside the control flow. Therefore, when users modify the network, they need to manually handle the node information inside and outside the control flow.

The current rewrite module has the following syntax limitations:

- Only networks of type *mindspore.nn.Cell* are supported as input to the rewrite module.
- Parsing one-line control flow syntax(e.g. one-line if-else, one-line for loop) is not currently supported.
- Parsing decorator syntax is not currently supported.
- Parsing local classes and embedded classes is not currently supported, that is, the definition of classes need to be placed on the outermost layer.

- Parsing closure syntax is not currently supported, that is, the definition of out-of-class functions need to be placed at the outermost layer.
- Parsing lambda expression syntax is not currently supported.
- Parsing global variables is not currently supported, that is, global variables need to be converted to class variables or local variables before they can be used.
- Parsing methods in the parent classes is not currently supported.

For statements that do not support parsing, rewrite will generate nodes of type *NodeType.Python* for corresponding statements to ensure that the network after rewrite can run normally. The *Python* node does not support modifying the input and output of statements, and there may be a problem between variable names and those generated by the rewrite. In this case, users need to adjust the variable names manually.

Parameters

network (Cell) – *network* used to create SymbolTree.

Returns

Symboltree, a SymbolTree created based on *network*.

Raises

TypeError – If *network* is not a *Cell* instance.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> print(type(stree))
<class 'mindspore.rewrite.api.symbol_tree.SymbolTree'>
```

erase(*node: Union[Node, str]*)

Erase a *node* from rewrite.

Parameters

node (*Union[Node, str]*) – A *Node* to be erased. Can be a node or name of node.

Returns

An instance of *Node* being erased if node is in *SymbolTree* else None.

Raises

TypeError – The type of *node* is not Node or str.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> stree.erase(node)
```

get_code()

Get source code corresponding to the network information in SymbolTree. If the network has already been modified, the source code of modified network is returned.

Returns

A str represents source code of modified network.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> codes = stree.get_code()
>>> print(codes)
```

get_network()

Get the network object generated based on SymbolTree. The source code is saved to a file in the 'rewritten_network' folder of the current directory.

Note:

- The modification of network by rewrite module is based on the modification of AST tree of original network instance, and the new network instance will obtain attribute information from original network instance, so the new network instance and the original network instance have data association, and the original network should no longer be used.
- Due to the data association between the new network and the original network instance, manually creating a network instance using the source code file generated by rewrite is not currently supported.

Returns

A network object generated from SymbolTree.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> new_net = stree.get_network()
```

get_node(*node_name: str*)

Get the node with the name *node_name* in the SymbolTree.

Parameters

node_name (*str*) – The name of node.

Returns

Node with name of *node_name* . Return `None` if there is no node named *node_name* in SymbolTree.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node('conv1')
>>> print(node.get_name())
conv1
```

insert(*position*, *node:* Node)

Insert a *node* into *SymbolTree* at *position*.

position is obtained from *before* api or *after* api of *SymbolTree*.

Parameters

- **position** (*Position*) – Indicate where to insert *node*.
- **node** (Node) – An instance of Node to be inserted.

Returns

An instance of Node being inserted.

Raises

- **ValueError** – If *position* is not belong to current *SymbolTree*.
- **TypeError** – If *position* is not a *Position*.
- **TypeError** – If *node* is not a *Node*.

Examples

```
>>> from mindspore.rewrite import SymbolTree, ScopedValue
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> position = stree.after(node)
>>> new_node = node.create_call_cell(cell=nn.ReLU(), targets=['x'],
...                                  args=[ScopedValue.create_naming_value('x')], name=
↪'new_relu')
>>> stree.insert(position, new_node)
```

nodes(*all_nodes: bool = False*)

Get the generator of the node in the current SymbolTree, which is used to iterate through the nodes in SymbolTree.

Parameters

all_nodes (*bool*) – Get all nodes including nodes in CallFunction node, CellContainer node and sub symbol tree. Default: `False` .

Returns

A generator for nodes in SymbolTree.

Raises

TypeError – If *all_nodes* is not bool.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> print([node.get_name() for node in stree.nodes()])
['input_x', 'conv1', 'relu', 'max_pool2d', 'conv2', 'relu_1', 'max_pool2d_1',
 'unaryop_not', 'if_node', 'flatten', 'fc1', 'relu_2', 'fc2', 'relu_3', 'fc3', 'return_1
↪']
```

print_node_tabulate(*all_nodes: bool = False*)

Print the topology information of nodes in SymbolTree, including node type, node name, node code, and node input-output relationship.

The information is output to the screen using the print interface, including the following information:

- **node type** (str): The type of node, refer to class:*mindspore.rewrite.NodeType* .
- **name** (str): The name of node.
- **codes** (str): The code statement in the SymbolTree corresponding to the node.
- **arg providers** (Dict[int, Tuple[str, int]]): The format is *{[idx, (n, k)]}* , which means the *idx* th parameter of the node is provided by the *k* th output of node *n* .
- **target users** (Dict[int, List[Tuple[str, int]]]): The format is '{[idx, [(n, k)]]}' , which means the *idx* th output of the node is used as the *k* th parameter of node *n* .

Parameters

all_nodes (*bool*) –Print information of all nodes, including nodes in CallFunction node, CellContainer node and sub symbol tree. Default: `False` .

Raises

TypeError –If *all_nodes* is not bool.

Examples

```
>>> from mindspore.rewrite import SymbolTree
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> stree.print_node_tabulate()
```

replace(*old_node:* Node, *new_nodes: List[*Node*]*)

Replace the *old_node* with nodes in the *new_nodes* list.

Nodes in *new_nodes* will be inserted into SymbolTree sequentially, and then *old_node* will be deleted.

Note:

- Replace only support one-to-one replacement or one-to-multi replacement.
- Caller should maintain the topological relationship between each node in the *new_nodes* , as well as the topological relationship between nodes in the *new_nodes* and nodes in the original tree.

Parameters

- **old_node** (Node) –Node to be replaced.
- **new_nodes** (*List[*Node*]*) –Nodes of the node_tree to replace in.

Returns

An instance of Node represents root of node_tree been replaced in.

Raises

- **TypeError** –If *old_node* is not a *Node*.
- **TypeError** –If *new_nodes* is not a *list* or node in *new_nodes* is not a *Node*.

Examples

```
>>> from mindspore.rewrite import SymbolTree, ScopedValue
>>> import mindspore.nn as nn
>>> # Define the network structure of LeNet5. Refer to
>>> # https://gitee.com/mindspore/docs/blob/r2.6.0/docs/mindspore/code/lenet.py
>>> net = LeNet5()
>>> stree = SymbolTree.create(net)
>>> node = stree.get_node("conv1")
>>> new_node = node.create_call_cell(cell=nn.ReLU(), targets=['x'],
```

(continues on next page)

(continued from previous page)

```
...                                     args=[ScopedValue.create_naming_value('x')], name=
↪'new_relu')
>>> stree.replace(node, [new_node])
```

unique_name(*name: str = 'output'*)

Based on the given *name* , returns a new name that is unique within the symbol tree. This interface can be used when a variable name that does not conflict is required.

Parameters

name (*str, optional*) – The prefix of the name. Defaults to `"output"` .

Returns

str, A new, unique name within a symbol tree in the format *name_n*, where *n* is a numeric subscript. If there is no name conflict when entered *name*, there is no numeric subscript.

Raises

TypeError – The type of *name* is not str.

class mindspore.rewrite.**ValueType**

ValueType represents type of *ScopedValue*.

- A *NamingValue* represents a reference to another variable.
- A *CustomObjValue* represents an instance of custom class or an object whose type is out of range of base-type and container-type of ValueType.

CHAPTER

TWENTYTWO

MINDSPORE.HAL

Hal encapsulates interfaces for device, stream, event, and memory. MindSpore abstracts the corresponding modules from different backends, allowing users to schedule hardware resources at the Python layer.

22.1 Device

mindspore.hal.device_count	Return device count of specified device, this api will be deprecated and removed in future versions, please use the api *mindspore.device_context.cpu.device_count()*, *mindspore.device_context.gpu.device_count()*, *mindspore.device_context.ascend.device_count()* instead.
mindspore.hal.get_arch_list	Get the architecture list this MindSpore was compiled for, this api will be deprecated and removed in future versions.
mindspore.hal.get_device_capability	Get specified device's capability, this api will be deprecated and removed in future versions.
mindspore.hal.get_device_name	Get specified device's name, this api will be deprecated and removed in future versions.
mindspore.hal.get_device_properties	Get specified device's properties, this api will be deprecated and removed in future versions.
mindspore.hal.is_available	Return whether specified device is available, this api will be deprecated and removed in future versions, please use the api *mindspore.device_context.cpu.is_available()*, *mindspore.device_context.gpu.is_available()*, *mindspore.device_context.ascend.is_available()* instead.
mindspore.hal.is_initialized	Return whether specified device is initialized, this api will be deprecated and removed in future versions.

22.1.1 mindspore.hal.device_count

mindspore.hal.**device_count**(*device_target=None*)

Return device count of specified device, this api will be deprecated and removed in future versions, please use the api *mindspore.device_context.cpu.device_count()* , *mindspore.device_context.gpu.device_count()*, *mindspore.device_context.ascend.device_count()* instead.

Note: For CPU device, this method always returns 1.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

int

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.device_count(device_target))
```

22.1.2 mindspore.hal.get_arch_list

mindspore.hal.**get_arch_list**(*device_target=None*)

Get the architecture list this MindSpore was compiled for, this api will be deprecated and removed in future versions.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

str for GPU. None for Ascend and CPU.

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.get_arch_list(device_target))
```

22.1.3 mindspore.hal.get_device_capability

mindspore.hal.**get_device_capability**(*device_id*, *device_target=None*)

Get specified device's capability, this api will be deprecated and removed in future versions.

Parameters

- **device_id** (*int*) −The device id of which the capability will be returned.
- **device_target** (*str, optional*) −The target device specified, should be one of "CPU" , "GPU" and "Ascend" . Default None , represents the current device set by context.

Returns

tuple(param1, param2) for GPU.

- param1 - int, cuda major revision number.
- param2 - int, cuda minor revision number.

None for Ascend and CPU.

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.get_device_capability(0, device_target))
```

22.1.4 mindspore.hal.get_device_name

mindspore.hal.**get_device_name**(*device_id*, *device_target=None*)

Get specified device's name, this api will be deprecated and removed in future versions.

Note: This method always returns "CPU" for CPU device.

Parameters

- **device_id** (*int*) −The device id of which the name will be returned.
- **device_target** (*str, optional*) −The target device specified, should be one of "CPU" , "GPU" and "Ascend" . Default None , represents the current device set by context.

Returns

str

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.get_device_name(0, device_target))
```

22.1.5 mindspore.hal.get_device_properties

mindspore.hal.**get_device_properties**(*device_id*, *device_target=None*)

Get specified device's properties, this api will be deprecated and removed in future versions.

Note: For Ascend, device must be initialized before calling this method, or *total_memory* and *free_memory* will be 0, and *device_id* will be ignored since this method only returns current device's properties.

Parameters

- **device_id** (*int*) –The device id of which the properties will be returned.
- **device_target** (*str, optional*) –The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

- *cudaDeviceProp* for GPU.

```
cudaDeviceProp {
    name(str),
    major(int),
    minor(int),
    is_multi_gpu_board(int),
    is_integrated(int),
    multi_processor_count(int),
    total_memory(int),
    warp_size(int)
}
```

- *AscendDeviceProperties* for Ascend.

```
AscendDeviceProperties {
    name(str),
    total_memory(int),
    free_memory(int)
}
```

- None for CPU.

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.get_device_properties(0, device_target))
```

22.1.6 mindspore.hal.is_available

mindspore.hal.**is_available**(*device_target*)

Return whether specified device is available, this api will be deprecated and removed in future versions, please use the api *mindspore.device_context.cpu.is_available()* , *mindspore.device_context.gpu.is_available()*, *mindspore.device_context.ascend.is_available()* instead.

Parameters

device_target (*str*) –The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

bool

Examples

```
>>> import mindspore
>>> device_target = mindspore.context.get_context("device_target")
>>> print(mindspore.hal.is_available(device_target))
True
```

22.1.7 mindspore.hal.is_initialized

mindspore.hal.**is_initialized**(*device_target*)

Return whether specified device is initialized, this api will be deprecated and removed in future versions.

Note: MindSpore's devices "CPU", "GPU" and "Ascend" will be initialized in the following scenarios:

- For distributed job, device will be initialized after *mindspore.communication.init* method is called.
- For standalone job, device will be initialized after running the first operator or calling creating stream/event interfaces.

Parameters

device_target (*str*) –The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` .

Returns

bool

Examples

```
>>> import mindspore as ms
>>> import numpy as np
>>> from mindspore import Tensor, ops
>>> ms.context.set_context(device_target="CPU")
>>> assert not ms.hal.is_initialized("CPU")
>>> a = Tensor(np.ones([1, 2]), ms.float32)
>>> b = Tensor(np.ones([1, 2]), ms.float32)
>>> c = ops.add(a, b).asnumpy()
>>> print(ms.hal.is_initialized("CPU"))
True
```

22.2 Stream

mindspore.hal.communication_stream	Return communication stream on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.communication_stream()* instead.
mindspore.hal.current_stream	Return current stream used on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.current_stream()* instead.
mindspore.hal.default_stream	Return default stream on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.default_stream()* instead.
mindspore.hal.set_cur_stream	Set the current stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.set_cur_stream()* instead.
mindspore.hal.synchronize	Synchronize all streams on current device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.synchronize()* instead.
mindspore.hal.Stream	Wrapper around a device stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.Stream* instead.
mindspore.hal.StreamCtx	Context-manager that selects a given stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.StreamCtx* instead.

22.2.1 mindspore.hal.communication_stream

mindspore.hal.**communication_stream**()

Return communication stream on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.communication_stream()* instead.

Returns

stream (Stream), communication stream.

Examples

```
>>> import mindspore
>>> mindspore.hal.communication_stream()
Stream(device_name=Ascend, device_id:0, stream id:1)
```

22.2.2 mindspore.hal.current_stream

mindspore.hal.**current_stream**()

Return current stream used on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.current_stream()* instead.

Returns

stream (Stream), current stream.

Examples

```
>>> import mindspore
>>> cur_stream = mindspore.hal.current_stream()
>>> assert cur_stream == mindspore.hal.default_stream()
```

22.2.3 mindspore.hal.default_stream

mindspore.hal.**default_stream**()

Return default stream on this device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.default_stream()* instead.

Returns

stream (Stream), default stream.

Examples

```
>>> import mindspore
>>> cur_stream = mindspore.hal.current_stream()
>>> assert cur_stream == mindspore.hal.default_stream()
```

22.2.4 mindspore.hal.set_cur_stream

mindspore.hal.**set_cur_stream**(*stream*)

Set the current stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.set_cur_stream()* instead.

Parameters

stream (Stream) – selected stream. This function is a no-op if this argument is None.

Examples

```
>>> import mindspore
>>> cur_stream = mindspore.hal.current_stream()
>>> assert cur_stream == mindspore.hal.default_stream()
>>> s1 = mindspore.hal.Stream()
>>> mindspore.hal.set_cur_stream(s1)
>>> assert mindspore.hal.current_stream() == s1
>>> mindspore.hal.set_cur_stream(mindspore.hal.default_stream())
```

22.2.5 mindspore.hal.synchronize

mindspore.hal.**synchronize**()

Synchronize all streams on current device, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.synchronize()* instead.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1024, 2048]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2048, 4096]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
>>> mindspore.hal.synchronize()
>>> assert s1.query()
```

22.2.6 mindspore.hal.Stream

class mindspore.hal.**Stream**(*priority=0*, ***kwargs*)

Wrapper around a device stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.Stream* instead.

A device stream is a linear sequence of execution that belongs to a specific device, independent from other streams.

Parameters

- **priority** (*int, optional*) – priority of the stream, lower numbers represent higher priorities. By default, streams have priority 0.
- **kwargs** (*dict*) – keyword arguments.

query()

Check if all the work submitted has been completed.

Returns

A boolean indicating if all kernels in this stream are completed.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1024, 2048]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2048, 4096]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
>>> s1.synchronize()
>>> assert s1.query()
```

record_event(*event=None*)

Record an event.

Parameters

event (Event, *optional*) – event to record. If not given, a new one will be allocated.

Returns

Event, recorded event. If this argument is None, a new one will be allocated. Default is None.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([3, 3]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([3, 3]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = a + b
...     event = s1.record_event()
...     d = a * b
>>> cur_stream = mindspore.hal.current_stream()
>>> cur_stream.wait_event(event)
>>> e = c + 3
>>> print(e)
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

synchronize()

Wait for all the kernels in this stream to complete.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1024, 2048]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2048, 4096]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
>>> s1.synchronize()
>>> assert s1.query()
```

wait_event(*event*)

Make all future work submitted to the stream wait for an event.

Parameters

event (Event) – an event to wait for.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([3, 3]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([3, 3]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = a + b
...     event = s1.record_event()
...     d = a * b
>>> cur_stream = mindspore.hal.current_stream()
>>> cur_stream.wait_event(event)
>>> e = c + 3
>>> print(e)
[[5. 5. 5.]
 [5. 5. 5.]
 [5. 5. 5.]]
```

wait_stream(*stream*)

Synchronize with another stream.

All future work submitted to this stream will wait until all kernels submitted to a given stream at the time of call complete.

Parameters

stream (Stream) – a stream to synchronize.

Examples

```
>>> import mindspore
>>> s1 = mindspore.hal.Stream()
>>> s2 = mindspore.hal.Stream()
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
>>> with mindspore.hal.StreamCtx(s2):
...     s2.wait_stream(s1)
...     d = mindspore.ops.matmul(c, b)
>>> mindspore.hal.synchronize()
>>> print(d)
[[4. 4.]]
```

22.2.7 mindspore.hal.StreamCtx

class mindspore.hal.**StreamCtx**(*ctx_stream*)

Context-manager that selects a given stream, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.StreamCtx* instead.

All kernels queued within its context will be enqueued on a selected stream.

Parameters

ctx_stream (Stream) – selected stream. This manager is a no-op if it's None.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1024, 2048]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2048, 4096]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
>>> mindspore.hal.synchronize()
>>> assert s1.query()
```

22.3 Event

mindspore.hal.Event	Wrapper around a device event, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.Event()* instead.

22.3.1 mindspore.hal.Event

class mindspore.hal.**Event**(*enable_timing=False*, *blocking=False*)

Wrapper around a device event, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.Event()* instead.

Device events are synchronization markers that can be used to monitor the device' s progress, to accurately measure timing, and to synchronize device streams.

The underlying device events are lazily initialized when the event is first recorded.

Parameters

- **enable_timing** (*bool, optional*) – indicates if the event should measure time. Default False.
- **blocking** (*bool, optional*) – if True, *wait* will be blocking. Default False.

Examples

```
>>> import mindspore
>>> start = mindspore.hal.Event(enable_timing=True)
>>> end = mindspore.hal.Event(enable_timing=True)
>>> s1 = mindspore.hal.Stream()
>>> s2 = mindspore.hal.Stream()
>>> a = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> c = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> with mindspore.hal.StreamCtx(s1):
...     d = mindspore.ops.matmul(a, b)
...     start.record()
>>> c += 2
>>> end.record()
>>> with mindspore.hal.StreamCtx(s2):
...     start.synchronize()
...     end.synchronize()
...     e = c + d
>>> mindspore.hal.synchronize()
>>> print(e)
[[5. 5.]
 [5. 5.]]
>>> elapsed_time = start.elapsed_time(end)
```

elapsed_time(*end_event*)

Return the time elapsed in milliseconds after the event was recorded and before the end_event was recorded.

Parameters

end_event (Event) – end event.

Returns

float, the time elapsed in milliseconds.

query()

Check if all work currently captured by event has completed.

Returns

A boolean indicating if all work currently captured by event has completed.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1024, 2048]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2048, 4096]), mindspore.float32)
>>> s1 = mindspore.hal.Stream()
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
...     ev = s1.record_event()
>>> s1.synchronize()
>>> assert ev.query()
```

record(*stream=None*)

Record the event in a given stream.

Uses *mindspore.hal.current_stream()* if no *stream* is specified. The stream's device must match the event's device.

Parameters

stream (Stream, *optional*) – a stream to record. If this argument is None, current stream will be used. Default None.

synchronize()

Wait for the event to complete.

Waits until the completion of all work currently captured in this event. This prevents the CPU thread from proceeding until the event completes.

wait(*stream=None*)

Make all future work submitted to the given stream wait for this event.

Use *mindspore.hal.current_stream()* if no *stream* is specified.

Parameters

stream (Stream, *optional*) – a stream to record. If this argument is None, current stream will be used. Default None.

Examples

```
>>> import mindspore
>>> event = mindspore.hal.Event()
>>> s1 = mindspore.hal.Stream()
>>> s2 = mindspore.hal.Stream()
>>> a = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([2, 2]), mindspore.float32)
>>> with mindspore.hal.StreamCtx(s1):
...     c = mindspore.ops.matmul(a, b)
...     event.record()
>>> event.wait()
>>> d = c + 2
>>> mindspore.hal.synchronize()
>>> print(d)
[[4. 4.]
 [4. 4.]]
```

22.4 Memory

mindspore.hal.contiguous_tensors_handle.combine _tensor_list_contiguous	Return a contiguous memory handle where contiguous memory has been requested and slicing functionality is provided.
mindspore.hal.contiguous_tensors_handle.ContiguousTensorsHandle	ContiguousTensorsHandle is a handle manage continuous memory.
mindspore.hal.max_memory_allocated	Return the peak memory size of the memory pool actually occupied by Tensor since the process was started.
mindspore.hal.max_memory_reserved	Returns the peak value of the total memory managed by the memory pool since the process was started.

continues on next page

Table 4 – continued from previous page

mindspore.hal.memory_allocated	Returns the actual memory size currently occupied by Tensor, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.memory_allocated()* instead.
mindspore.hal.memory_reserved	Returns the total amount of memory currently managed by the memory pool, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.memory_reserved()* instead.
mindspore.hal.memory_stats	Returns status information queried from the memory pool, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.memory_stats()* instead.
mindspore.hal.memory_summary	Returns readable memory pool status information, this api will be deprecated and removed in future versions.
mindspore.hal.reset _max_memory_reserved	Reset the peak memory size managed by the memory pool, this api will be deprecated and removed in future versions.
mindspore.hal.reset _max_memory_allocated	Reset the peak memory size of the memory pool actually occupied by Tensor, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.reset_max_memory_allocated()* instead.
mindspore.hal.reset _peak_memory_stats	Reset the "peak" stats tracked by memory manager, this api will be deprecated and removed in future versions.
mindspore.hal.empty_cache	Empty cache in the memory pool, this api will be deprecated and removed in future versions.

22.4.1 mindspore.hal.contiguous_tensors_handle.combine_tensor_list_contiguous

mindspore.hal.contiguous_tensors_handle.**combine_tensor_list_contiguous**(*tensor_list*, *enable_mem_align=True*)

Return a contiguous memory handle where contiguous memory has been requested and slicing functionality is provided.

Parameters

- **tensor_list** (*list[Tensor], Tuple[Tensor]*) –The tensor list to be stored.
- **enable_mem_align** (*bool, optional*) –Whether to enable the memory alignment function. False is not supported. Default `True` .

Returns

ContiguousTensorsHandle, a manager with contiguous memory.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.hal.contiguous_tensors_handle import combine_tensor_list_contiguous
>>> x = Tensor(np.array([1, 2, 3]).astype(np.float32))
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> handle = combine_tensor_list_contiguous([x, y], True)
>>> print(handle[0].shape)
[1]
>>> print(handle[1: 3].asnumpy())
```

(continues on next page)

(continued from previous page)

```
[2, 3]
>>> print(output.slice_by_tensor_index(0, 1).asnumpy())
[1, 2, 3]
```

22.4.2 mindspore.hal.contiguous_tensors_handle.ContiguousTensorsHandle

class mindspore.hal.contiguous_tensors_handle.**ContiguousTensorsHandle**(*tensor_list*, *enable_mem_align=True*)

ContiguousTensorsHandle is a handle manage continuous memory.

Parameters

- **tensor_list** (*list[Tensor], Tuple[Tensor]*) –The tensor list to be stored.
- **enable_mem_align** (*bool, optional*) –Whether to enable the memory alignment function. False is not supported. Default `True` .

Returns

ContiguousTensorsHandle, a manager with contiguous memory.

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.hal.contiguous_tensors_handle import ContiguousTensorsHandle
>>> x = Tensor(np.array([1, 2, 3]).astype(np.float32))
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> handle = ContiguousTensorsHandle([x, y], True)
>>> print(handle[0].shape)
[1]
>>> print(handle[1: 3].asnumpy())
[2, 3]
```

slice_by_tensor_index(*start=None*, *end=None*)

Return the tensor which is sliced by tensor index.

Parameters

- **start** (*int, None*) –Starting position. Default `None`.
- **end** (*int, None*) –Deadline position. Default `None`.

Returns

Tensor

Examples

```
>>> import numpy as np
>>> import mindspore as ms
>>> from mindspore import Tensor
>>> from mindspore.hal.contiguous_tensors_handle import ContiguousTensorsHandle
>>> x = Tensor(np.array([1, 2, 3]).astype(np.float32))
>>> y = Tensor(np.array([4, 5, 6]).astype(np.float32))
>>> handle = ContiguousTensorsHandle([x, y], True)
>>> print(output.slice_by_tensor_index(0, 1).asnumpy())
[1, 2, 3]
```

22.4.3 mindspore.hal.max_memory_allocated

mindspore.hal.**max_memory_allocated**(*device_target=None*)

Return the peak memory size of the memory pool actually occupied by Tensor since the process was started. This api will be deprecated and removed in future versions, please use the api *mindspore.runtime.max_memory_allocated()* instead.

Note:

- For the *CPU* device, 0 is always returned.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

int, in Byte.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.max_memory_allocated())
1536
```

22.4.4 mindspore.hal.max_memory_reserved

mindspore.hal.**max_memory_reserved**(*device_target=None*)

Returns the peak value of the total memory managed by the memory pool since the process was started. This api will be deprecated and removed in future versions, please use the api *mindspore.runtime.max_memory_reserved()* instead.

Note:

- For the *CPU* device, 0 is always returned.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

int, in Byte.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.max_memory_reserved())
1073741824
```

22.4.5 mindspore.hal.memory_allocated

`mindspore.hal.`**`memory_allocated`**(*device_target=None*)

Returns the actual memory size currently occupied by Tensor, this api will be deprecated and removed in future versions, please use the api *`mindspore.runtime.memory_allocated()`* instead.

Note:

- For the *CPU* device, 0 is always returned.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

int, in Byte.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.memory_allocated())
1024
```

22.4.6 mindspore.hal.memory_reserved

mindspore.hal.**memory_reserved**(*device_target=None*)

Returns the total amount of memory currently managed by the memory pool, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.memory_reserved()* instead.

Note:

- For the *CPU* device, 0 is always returned.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

int, in Byte.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.memory_reserved())
1073741824
```

22.4.7 mindspore.hal.memory_stats

mindspore.hal.**memory_stats**(*device_target=None*)

Returns status information queried from the memory pool, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.memory_stats()* instead.

Note:

- For the *CPU* device, a dictionary with empty data is always returned.

Parameters

device_target (*str, optional*) – The target device specified, should be one of `"CPU"` , `"GPU"` and `"Ascend"` . Default `None` , represents the current device set by context.

Returns

dict, the queried memory information.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.memory_stats())
{'total_reserved_memory': 1073741824, 'total_allocated_memory': 1024, 'total_idle_memory':␣
↪1073740800,
'total_eager_free_memory': 0, 'max_reserved_memory': 1073741824, 'max_allocated_memory':␣
↪1536,
'common_mem_pool_stats': {'block_unit_size': 1073741824, 'block_counts': 1, 'blocks_info':
{<capsule object NULL at 0x7f7e8c27b030>: {'block_stream_id': 0, 'block_memory_size':␣
↪1073741824}}},
'persistent_mem_pool_stats': {'block_unit_size': 1073741824, 'block_counts': 0, 'blocks_info
↪': {}}}
```

22.4.8 mindspore.hal.memory_summary

mindspore.hal.**memory_summary**(*device_target=None*)

Returns readable memory pool status information, this api will be deprecated and removed in future versions. Please use the api *mindspore.runtime.memory_summary()* instead.

Parameters

device_target (*str, optional*) – The target device specified, should be one of "CPU", "GPU" and "Ascend". Default None, represents the current device set by context.

Returns

str, readable memory pool status information in tabular form.

22.4.9 mindspore.hal.reset_max_memory_reserved

mindspore.hal.**reset_max_memory_reserved**(*device_target=None*)

Reset the peak memory size managed by the memory pool, this api will be deprecated and removed in future versions. Please use the api *mindspore.runtime.reset _max_memory_reserved()* instead.

Parameters

device_target (*str, optional*) – The target device specified, should be one of "CPU", "GPU" and "Ascend". Default None, represents the current device set by context.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.max_memory_reserved())
1073741824
>>> mindspore.hal.reset_max_memory_reserved()
>>> print(mindspore.hal.max_memory_reserved())
0
```

22.4.10 mindspore.hal.reset_max_memory_allocated

mindspore.hal.**reset_max_memory_allocated**(*device_target=None*)

Reset the peak memory size of the memory pool actually occupied by Tensor, this api will be deprecated and removed in future versions, please use the api *mindspore.runtime.reset _max_memory_allocated()* instead.

Parameters

device_target (*str, optional*) – The target device specified, should be one of "CPU", "GPU" and "Ascend". Default None, represents the current device set by context.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.max_memory_allocated())
1536
>>> mindspore.hal.reset_max_memory_allocated()
>>> print(mindspore.hal.max_memory_allocated())
0
```

22.4.11 mindspore.hal.reset_peak_memory_stats

mindspore.hal.**reset_peak_memory_stats**(*device_target=None*)

Reset the "peak" stats tracked by memory manager, this api will be deprecated and removed in future versions. Please use the api *mindspore.runtime.reset _peak_memory_stats()* instead.

Parameters

device_target (*str, optional*) – The target device specified, should be one of "CPU", "GPU" and "Ascend". Default None, represents the current device set by context.

Examples

```
>>> import mindspore
>>> a = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> b = mindspore.tensor(mindspore.ops.ones([1, 2]), mindspore.float32)
>>> c = mindspore.ops.add(a, b).asnumpy()
>>> print(mindspore.hal.max_memory_reserved())
1073741824
>>> print(mindspore.hal.max_memory_allocated())
1536
>>> mindspore.hal.reset_peak_memory_stats()
>>> print(mindspore.hal.max_memory_reserved())
0
>>> print(mindspore.hal.max_memory_allocated())
0
```

22.4.12 mindspore.hal.empty_cache

mindspore.hal.**empty_cache**()

Empty cache in the memory pool, this api will be deprecated and removed in future versions. Please use the api *mindspore.runtime.empty_cache()* instead.

Note:

- Empty cache help reduce the fragmentation of device memory.
- Support Atlas A2 series products.

Supported Platforms:

`Ascend`