tf.Variable

Contents

Class Variable
Child Classes
Properties
constraint

Class `Variable`

Defined in `tensorflow/python/ops/variables.py`.

See the guide: Variables > Variables

See the Variables How To for a high level overview.

A variable maintains state in the graph across calls to `run()`. You add a variable to the graph by constructing an instance of the class `Variable`.

The `Variable()` constructor requires an initial value for the variable, which can be a `Tensor` of any type and shape. The initial value defines the type and shape of the variable. After construction, the type and shape of the variable are fixed. The value can be changed using one of the assign methods.

If you want to change the shape of a variable later you have to use an `assign` Op with `validate_shape=False`.

Just like any `Tensor`, variables created with `Variable()` can be used as inputs for other Ops in the graph. Additionally, all the operators overloaded for the `Tensor` class are carried over to variables, so you can also add nodes to the graph by just doing arithmetic on variables.

```
import tensorflow as tf

# Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

# Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

# The overloaded operators are available too.
z = tf.sigmoid(w + y)

# Assign a new value to the variable with `assign()` or a related method.
w.assign(w + 1.0)
w.assign_add(1.0)
```

When you launch the graph, variables have to be explicitly initialized before you can run Ops that use their value. You can initialize a variable by running its *initializer op*, restoring the variable from a save file, or simply running an `assign` Op that assigns a value to the variable. In fact, the variable *initializer op* is just an `assign` Op that assigns the variable's initial value to the variable itself.

```
# Launch the graph in a session.
with tf.Session() as sess:
    # Run the variable initializer.
    sess.run(w.initializer)
    # ...you now can run ops that use the value of 'w'...
```

The most common initialization pattern is to use the convenience function `global_variables_initializer()` to add an Op to the graph that initializes all the variables. You then run that Op after launching the graph.

```
# Add an Op to initialize global variables.
init_op = tf.global_variables_initializer()

# Launch the graph in a session.
with tf.Session() as sess:
    # Run the Op that initializes global variables.
    sess.run(init_op)
    # ...you can now run any Op that uses variable values...
```

If you need to create a variable with an initial value dependent on another variable, use the other variable's `initialized_value()`. This ensures that variables are initialized in the right order.

All variables are automatically collected in the graph where they are created. By default, the constructor adds the new variable to the graph collection `GraphKeys.GLOBAL_VARIABLES`. The convenience function `global_variables()` returns the contents of that collection.

When building a machine learning model it is often convenient to distinguish between variables holding the trainable model parameters and other variables such as a `global step` variable used to count training steps. To make this easier, the variable constructor supports a `trainable=<bool>` parameter. If `True`, the new variable is also added to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. The convenience function `trainable_variables()` returns the contents of this collection. The various `Optimizer` classes use this collection as the default list of variables to optimize.

Child Classes

`class SaveSliceInfo`

Properties

`constraint`

Returns the constraint function associated with this variable.

Returns:

The constraint function that was passed to the variable constructor. Can be `None` if no constraint was passed.

`device`

The device of this variable.

`dtype`

The `DType` of this variable.

graph

The `Graph` of this variable.

initial_value

Returns the Tensor used as the initial value for the variable.

Note that this is different from `initialized_value()` which runs the op that initializes the variable before returning its value. This method returns the tensor that is used by the op that initializes the variable.

Returns:

A `Tensor`.

initializer

The initializer operation for this variable.

name

The name of this variable.

op

The `Operation` of this variable.

shape

The `TensorShape` of this variable.

Returns:

A `TensorShape`.

Methods

__init__

```
__init__(
    initial_value=None,
    trainable=True,
    collections=None,
    validate_shape=True,
    caching_device=None,
    name=None,
    variable_def=None,
    dtype=None,
    expected_shape=None,
    import_scope=None,
    constraint=None
)
```

Creates a new variable with value `initial_value`.

The new variable is added to the graph collections listed in `collections`, which defaults to `[GraphKeys.GLOBAL_VARIABLES]`.

If `trainable` is `True` the variable is also added to the graph collection `GraphKeys.TRAINABLE_VARIABLES`.

This constructor creates both a `variable` Op and an `assign` Op to set the variable to its initial value.

Args:

- `initial_value`: A `Tensor`, or Python object convertible to a `Tensor`, which is the initial value for the Variable. The initial value must have a shape specified unless `validate_shape` is set to False. Can also be a callable with no argument that returns the initial value when called. In that case, `dtype` must be specified. (Note that initializer functions from init_ops.py must first be bound to a shape before being used here.)
- `trainable`: If `True`, the default, also adds the variable to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. This collection is used as the default list of variables to use by the `Optimizer` classes.
- `collections`: List of graph collections keys. The new variable is added to these collections. Defaults to `[GraphKeys.GLOBAL_VARIABLES]`.
- `validate_shape`: If `False`, allows the variable to be initialized with a value of unknown shape. If `True`, the default, the shape of `initial_value` must be known.
- `caching_device`: Optional device string describing where the Variable should be cached for reading. Defaults to the Variable's device. If not `None`, caches on another device. Typical use is to cache on the device where the Ops using the Variable reside, to deduplicate copying through `Switch` and other conditional statements.
- `name`: Optional name for the variable. Defaults to `'Variable'` and gets uniquified automatically.
- `variable_def`: `VariableDef` protocol buffer. If not `None`, recreates the Variable object with its contents, referencing the variable's nodes in the graph, which must already exist. The graph is not changed. `variable_def` and the other arguments are mutually exclusive.
- `dtype`: If set, initial_value will be converted to the given type. If `None`, either the datatype will be kept (if `initial_value` is a Tensor), or `convert_to_tensor` will decide.
- `expected_shape`: A TensorShape. If set, initial_value is expected to have this shape.
- `import_scope`: Optional `string`. Name scope to add to the `Variable.` Only used when initializing from protocol buffer.
- `constraint`: An optional projection function to be applied to the variable after being updated by an `Optimizer` (e.g. used to implement norm constraints or value constraints for layer weights). The function must take as input the unprojected Tensor representing the value of the variable and return the Tensor for the projected value (which must have the same shape). Constraints are not safe to use when doing asynchronous distributed training.

Raises:

- `ValueError`: If both `variable_def` and initial_value are specified.
- `ValueError`: If the initial value is not specified, or does not have a shape and `validate_shape` is `True`.
- `RuntimeError`: If created in EAGER mode.

__abs__

```
__abs__(
    a,
    *args
)
```

Computes the absolute value of a tensor.

Given a tensor `x` of complex numbers, this operation returns a tensor of type `float32` or `float64` that is the absolute value of each element in `x`. All elements in `x` must be complex numbers of the form *a* + *bj*. The absolute value is computed as $\sqrt{a^2 + b^2}$. For example:

```
x = tf.constant([[-2.25 + 4.75j], [-3.25 + 5.75j]])
tf.abs(x)  # [5.25594902, 6.60492229]
```

Args:

- `x`: A `Tensor` or `SparseTensor` of type `float32`, `float64`, `int32`, `int64`, `complex64` or `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` the same size and type as `x` with absolute values. Note, for `complex64` or `complex128' input, the returned` Tensor `will be of type` float32 `or` float64`, respectively.

__add__

```
__add__(
    a,
    *args
)
```

Returns x + y element-wise.

NOTE: `Add` supports broadcasting. `AddN` does not. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `string`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__and__

```
__and__(
    a,
    *args
)
```

Returns the truth value of x AND y element-wise.

NOTE: `LogicalAnd` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__div__`

```
__div__(
    a,
    *args
)
```

Divide two values using Python 2 semantics. Used for Tensor.**div**.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` returns the quotient of x and y.

`__floordiv__`

```
__floordiv__(
    a,
    *args
)
```

Divides `x / y` elementwise, rounding toward the most negative integer.

The same as `tf.div(x,y)` for integers, but uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).

`x` and `y` must have the same type, and the result will have the same type as well.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

Raises:

- `TypeError`: If the inputs are complex.

__ge__

```
__ge__(
    a,
    *args
)
```

Returns the truth value of (x >= y) element-wise.

NOTE: `GreaterEqual` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

__getitem__

```
__getitem__(
    var,
    slice_spec
)
```

Creates a slice helper object given a variable.

This allows creating a sub-tensor from part of the current contents of a variable. See ${tf.Tensor$ `Tensor.__getitem__` } for detailed examples of slicing.

This function in addition also allows assignment to a sliced range. This is similar to `__setitem__` functionality in Python. However, the syntax is different so that the user can capture the assignment operation for grouping or passing to `sess.run()`. For example,

```
import tensorflow as tf
A = tf.Variable([[1,2,3], [4,5,6], [7,8,9]], dtype=tf.float32)
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  print(sess.run(A[:2, :2]))  # => [[1,2], [4,5]]

  op = A[:2,:2].assign(22. * tf.ones((2, 2)))
  print(sess.run(op))  # => [[22, 22, 3], [22, 22, 6], [7,8,9]]
```

Note that assignments currently do not support NumPy broadcasting semantics.

Args:

- `var`: An `ops.Variable` object.
- `slice_spec`: The arguments to `Tensor.__getitem__`.

Returns:

The appropriate slice of "tensor", based on "slice_spec". As an operator. The operator also has a `assign()` method that can be used to generate an assignment operator.

Raises:

- `ValueError`: If a slice range is negative size.
- `TypeError`: If the slice indices aren't int, slice, or Ellipsis.

`__gt__`

```
__gt__(
    a,
    *args
)
```

Returns the truth value of (x > y) element-wise.

NOTE: `Greater` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__invert__`

```
__invert__(
    a,
    *args
)
```

Returns the truth value of NOT x element-wise.

Args:

- `x`: A `Tensor` of type `bool`.

- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__iter__`

```
__iter__()
```

Dummy method to prevent iteration. Do not call.

NOTE(mrry): If we register **getitem** as an overloaded operator, Python will valiantly attempt to iterate over the variable's Tensor from 0 to infinity. Declaring this method prevents this unintended behavior.

Raises:

- `TypeError`: when invoked.

`__le__`

```
__le__(
    a,
    *args
)
```

Returns the truth value of (x <= y) element-wise.

NOTE: `LessEqual` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__lt__`

```
__lt__(
    a,
    *args
)
```

Returns the truth value of (x < y) element-wise.

NOTE: `Less` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__matmul__`

```
__matmul__(
    a,
    *args
)
```

Multiplies matrix `a` by matrix `b`, producing `a` * `b`.

The inputs must, following any transpositions, be tensors of rank >= 2 where the inner 2 dimensions specify valid matrix multiplication arguments, and any further outer dimensions match.

Both matrices must be of the same type. The supported types are: `float16`, `float32`, `float64`, `int32`, `complex64`, `complex128`.

Either matrix can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to `True`. These are `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default. This optimization is only available for plain matrices (rank-2 tensors) with datatypes `bfloat16` or `float32`.

For example:

```
# 2-D tensor `a`
# [[1, 2, 3],
#  [4, 5, 6]]
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])

# 2-D tensor `b`
# [[ 7,  8],
#  [ 9, 10],
#  [11, 12]]
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])

# `a` * `b`
# [[ 58,  64],
#  [139, 154]]
c = tf.matmul(a, b)

# 3-D tensor `a`
# [[[ 1,  2,  3],
#   [ 4,  5,  6]],
#  [[ 7,  8,  9],
#   [10, 11, 12]]]
a = tf.constant(np.arange(1, 13, dtype=np.int32),
                shape=[2, 2, 3])

# 3-D tensor `b`
# [[[13, 14],
#   [15, 16],
#   [17, 18]],
#  [[19, 20],
#   [21, 22],
#   [23, 24]]]
b = tf.constant(np.arange(13, 25, dtype=np.int32),
                shape=[2, 3, 2])

# `a` * `b`
# [[[ 94, 100],
#   [229, 244]],
#  [[508, 532],
#   [697, 730]]]
c = tf.matmul(a, b)

# Since python >= 3.5 the @ operator is supported (see PEP 465).
# In TensorFlow, it simply calls the `tf.matmul()` function, so the
# following lines are equivalent:
d = a @ b @ [[10.], [11.]]
d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])
```

Args:

- `a`: **`Tensor`** of type **`float16`**, **`float32`**, **`float64`**, **`int32`**, **`complex64`**, **`complex128`** and rank > 1.
- `b`: **`Tensor`** with same type and rank as **`a`**.
- `transpose_a`: If **`True`**, **`a`** is transposed before multiplication.
- `transpose_b`: If **`True`**, **`b`** is transposed before multiplication.
- `adjoint_a`: If **`True`**, **`a`** is conjugated and transposed before multiplication.
- `adjoint_b`: If **`True`**, **`b`** is conjugated and transposed before multiplication.
- `a_is_sparse`: If **`True`**, **`a`** is treated as a sparse matrix.
- `b_is_sparse`: If **`True`**, **`b`** is treated as a sparse matrix.
- `name`: Name for the operation (optional).

Returns:

A `Tensor` of the same type as `a` and `b` where each inner-most matrix is the product of the corresponding matrices in `a` and `b`, e.g. if all transpose or adjoint attributes are `False`:

`output`[..., i, j] = sum_k (`a`[..., i, k] * `b`[..., k, j]), for all indices i, j.

- `Note`: This is matrix product, not element-wise product.

Raises:

- `ValueError`: If transpose_a and adjoint_a, or transpose_b and adjoint_b are both set to True.

__mod__

```
__mod__(
    a,
    *args
)
```

Returns element-wise remainder of division. When `x < 0` xor `y < 0` is

true, this follows Python semantics in that the result here is consistent with a flooring divide. E.g. `floor(x / y) * y + mod(x, y) = x`.

NOTE: `FloorMod` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `int32`, `int64`, `float32`, `float64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__mul__

```
__mul__(
    a,
    *args
)
```

Dispatches cwise mul for "Dense*Dense*" *and* "*Dense*Sparse".

__neg__

```
__neg__(
    a,
    *args
)
```

Computes numerical negative value element-wise.

I.e., $y = -x$.

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__or__`

```
__or__(
    a,
    *args
)
```

Returns the truth value of x OR y element-wise.

NOTE: `LogicalOr` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__pow__`

```
__pow__(
    a,
    *args
)
```

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes x^y for corresponding elements in `x` and `y`. For example:

```
x = tf.constant([[2, 2], [3, 3]])
y = tf.constant([[8, 16], [2, 3]])
tf.pow(x, y)  # [[256, 65536], [9, 27]]
```

Args:

- `x`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `y`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.

- `name`: A name for the operation (optional).

Returns:

A `Tensor`.

`__radd__`

```
__radd__(
    a,
    *args
)
```

Returns x + y element-wise.

NOTE: `Add` supports broadcasting. `AddN` does not. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `string`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__rand__`

```
__rand__(
    a,
    *args
)
```

Returns the truth value of x AND y element-wise.

NOTE: `LogicalAnd` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__rdiv__`

```
__rdiv__(
    a,
    *args
)
```

Divide two values using Python 2 semantics. Used for Tensor.**div**.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` returns the quotient of x and y.

`__rfloordiv__`

```
__rfloordiv__(
    a,
    *args
)
```

Divides `x / y` elementwise, rounding toward the most negative integer.

The same as `tf.div(x,y)` for integers, but uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).

`x` and `y` must have the same type, and the result will have the same type as well.

Args:

- `x`: `Tensor` numerator of real numeric type.
- `y`: `Tensor` denominator of real numeric type.
- `name`: A name for the operation (optional).

Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

Raises:

- `TypeError`: If the inputs are complex.

`__rmatmul__`

```
__rmatmul__(
    a,
    *args
)
```

Multiplies matrix `a` by matrix `b`, producing `a` * `b`.

The inputs must, following any transpositions, be tensors of rank >= 2 where the inner 2 dimensions specify valid matrix multiplication arguments, and any further outer dimensions match.

Both matrices must be of the same type. The supported types are: `float16`, `float32`, `float64`, `int32`, `complex64`, `complex128`.

Either matrix can be transposed or adjointed (conjugated and transposed) on the fly by setting one of the corresponding flag to `True`. These are `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default. This optimization is only available for plain matrices (rank-2 tensors) with datatypes `bfloat16` or `float32`.

For example:

```
# 2-D tensor `a`
# [[1, 2, 3],
#  [4, 5, 6]]
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])

# 2-D tensor `b`
# [[ 7,  8],
#  [ 9, 10],
#  [11, 12]]
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])

# `a` * `b`
# [[ 58,  64],
#  [139, 154]]
c = tf.matmul(a, b)

# 3-D tensor `a`
# [[[ 1,  2,  3],
#   [ 4,  5,  6]],
#  [[ 7,  8,  9],
#   [10, 11, 12]]]
a = tf.constant(np.arange(1, 13, dtype=np.int32),
                shape=[2, 2, 3])

# 3-D tensor `b`
# [[[13, 14],
#   [15, 16],
#   [17, 18]],
#  [[19, 20],
#   [21, 22],
#   [23, 24]]]
b = tf.constant(np.arange(13, 25, dtype=np.int32),
                shape=[2, 3, 2])

# `a` * `b`
# [[[ 94, 100],
#   [229, 244]],
#  [[508, 532],
#   [697, 730]]]
c = tf.matmul(a, b)

# Since python >= 3.5 the @ operator is supported (see PEP 465).
# In TensorFlow, it simply calls the `tf.matmul()` function, so the
# following lines are equivalent:
d = a @ b @ [[10.], [11.]]
d = tf.matmul(tf.matmul(a, b), [[10.], [11.]])
```

Args:

- `a`: **`Tensor`** of type **`float16`**, **`float32`**, **`float64`**, **`int32`**, **`complex64`**, **`complex128`** and rank > 1.
- `b`: **`Tensor`** with same type and rank as **`a`**.
- `transpose_a`: If **`True`**, **`a`** is transposed before multiplication.
- `transpose_b`: If **`True`**, **`b`** is transposed before multiplication.
- `adjoint_a`: If **`True`**, **`a`** is conjugated and transposed before multiplication.
- `adjoint_b`: If **`True`**, **`b`** is conjugated and transposed before multiplication.
- `a_is_sparse`: If **`True`**, **`a`** is treated as a sparse matrix.
- `b_is_sparse`: If **`True`**, **`b`** is treated as a sparse matrix.
- `name`: Name for the operation (optional).

Returns:

A `Tensor` of the same type as `a` and `b` where each inner-most matrix is the product of the corresponding matrices in `a` and `b`, e.g. if all transpose or adjoint attributes are `False`:

`output`[..., i, j] = sum_k (`a`[..., i, k] * `b`[..., k, j]), for all indices i, j.

- `Note`: This is matrix product, not element-wise product.

Raises:

- `ValueError`: If transpose_a and adjoint_a, or transpose_b and adjoint_b are both set to True.

__rmod__

```
__rmod__(
    a,
    *args
)
```

Returns element-wise remainder of division. When `x < 0` xor `y < 0` is

true, this follows Python semantics in that the result here is consistent with a flooring divide. E.g. `floor(x / y) * y + mod(x, y) = x`.

NOTE: `FloorMod` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `int32`, `int64`, `float32`, `float64`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

__rmul__

```
__rmul__(
    a,
    *args
)
```

Dispatches cwise mul for "Dense*Dense*" *and* "*Dense*Sparse".

__ror__

```
__ror__(
    a,
    *args
)
```

Returns the truth value of x OR y element-wise.

NOTE: `LogicalOr` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor` of type `bool`.
- `y`: A `Tensor` of type `bool`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

`__rpow__`

```
__rpow__(
    a,
    *args
)
```

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes x^y for corresponding elements in `x` and `y`. For example:

```
x = tf.constant([[2, 2], [3, 3]])
y = tf.constant([[8, 16], [2, 3]])
tf.pow(x, y)  # [[256, 65536], [9, 27]]
```

Args:

- `x`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `y`: A `Tensor` of type `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`.

`__rsub__`

```
__rsub__(
    a,
    *args
)
```

Returns x - y element-wise.

NOTE: `Sub` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `uint16`, `int16`, `int32`, `int64`, `complex64`, `complex128`.

- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__rtruediv__`

```
__rtruediv__(
    a,
    *args
)
```

`__rxor__`

```
__rxor__(
    a,
    *args
)
```

x ^ y = (x | y) & ~(x & y).

`__sub__`

```
__sub__(
    a,
    *args
)
```

Returns x - y element-wise.

NOTE: `Sub` supports broadcasting. More about broadcasting here

Args:

- `x`: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `uint16`, `int16`, `int32`, `int64`, `complex64`, `complex128`.
- `y`: A `Tensor`. Must have the same type as `x`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.

`__truediv__`

```
__truediv__(
    a,
    *args
)
```

`__xor__`

```
__xor__(
    a,
    *args
)
```

x ^ y = (x | y) & ~(x & y).

`assign`

```
assign(
    value,
    use_locking=False
)
```

Assigns a new value to the variable.

This is essentially a shortcut for `assign(self, value)`.

Args:

- `value`: A `Tensor`. The new value for this variable.
- `use_locking`: If `True`, use locking during the assignment.

Returns:

A `Tensor` that will hold the new value of this variable after the assignment has completed.

`assign_add`

```
assign_add(
    delta,
    use_locking=False
)
```

Adds a value to this variable.

This is essentially a shortcut for `assign_add(self, delta)`.

Args:

- `delta`: A `Tensor`. The value to add to this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the addition has completed.

`assign_sub`

```
assign_sub(
    delta,
    use_locking=False
)
```

Subtracts a value from this variable.

This is essentially a shortcut for `assign_sub(self, delta)`.

Args:

- `delta`: A `Tensor`. The value to subtract from this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the subtraction has completed.

count_up_to

```
count_up_to(limit)
```

Increments this variable until it reaches `limit`.

When that Op is run it tries to increment the variable by `1`. If incrementing the variable would bring it above `limit` then the Op raises the exception `OutOfRangeError`.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for `count_up_to(self, limit)`.

Args:

- `limit`: value at which incrementing the variable raises an error.

Returns:

A `Tensor` that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

eval

```
eval(session=None)
```

In a session, computes and returns the value of this variable.

This is not a graph construction method, it does not add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See `tf.Session` for more information on launching a graph and on sessions.

```
v = tf.Variable([1, 2])
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    # Usage passing the session explicitly.
    print(v.eval(sess))
    # Usage with the default session.  The 'with' block
    # above makes 'sess' the default session.
    print(v.eval())
```

Args:

- `session`: The session to use to evaluate this variable. If none, the default session is used.

Returns:

A numpy `ndarray` with a copy of the value of this variable.

from_proto

```
@staticmethod
from_proto(
    variable_def,
    import_scope=None
)
```

Returns a `Variable` object created from `variable_def`.

get_shape

```
get_shape()
```

Alias of Variable.shape.

initialized_value

```
initialized_value()
```

Returns the value of the initialized variable.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
# Initialize 'v' with a random tensor.
v = tf.Variable(tf.truncated_normal([10, 40]))
# Use `initialized_value` to guarantee that `v` has been
# initialized before its value is used to initialize `w`.
# The random values are picked only once.
w = tf.Variable(v.initialized_value() * 2.0)
```

Returns:

A `Tensor` holding the value of this variable after its initializer has run.

load

```
load(
    value,
    session=None
)
```

Load new value into this variable.

Writes new value to variable's memory. Doesn't add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See `tf.Session` for more information on launching a graph and on sessions.

```
v = tf.Variable([1, 2])
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    # Usage passing the session explicitly.
    v.load([2, 3], sess)
    print(v.eval(sess)) # prints [2 3]
    # Usage with the default session.  The 'with' block
    # above makes 'sess' the default session.
    v.load([3, 4], sess)
    print(v.eval()) # prints [3 4]
```

Args:

- `value`: New variable value
- `session`: The session to use to evaluate this variable. If none, the default session is used.

Raises:

- `ValueError`: Session is not passed and no default session

read_value

```
read_value()
```

Returns the value of this variable, read in the current context.

Can be different from value() if it's on another device, with control dependencies, etc.

Returns:

A `Tensor` containing the value of the variable.

scatter_sub

```
scatter_sub(
    sparse_delta,
    use_locking=False
)
```

Subtracts `IndexedSlices` from this variable.

This is essentially a shortcut for `scatter_sub(self, sparse_delta.indices, sparse_delta.values)`.

Args:

- `sparse_delta`: `IndexedSlices` to be subtracted from this variable.
- `use_locking`: If `True`, use locking during the operation.

Returns:

A `Tensor` that will hold the new value of this variable after the scattered subtraction has completed.

Raises:

- `ValueError`: if `sparse_delta` is not an `IndexedSlices`.

`set_shape`

```
set_shape(shape)
```

Overrides the shape for this variable.

Args:

- `shape`: the `TensorShape` representing the overridden shape.

`to_proto`

```
to_proto(export_scope=None)
```

Converts a `Variable` to a `VariableDef` protocol buffer.

Args:

- `export_scope`: Optional `string`. Name scope to remove.

Returns:

A `VariableDef` protocol buffer, or `None` if the `Variable` is not in the specified name scope.

`value`

```
value()
```

Returns the last snapshot of this variable.

You usually do not need to call this method as all ops that need the value of the variable call it automatically through a `convert_to_tensor()` call.

Returns a `Tensor` which holds the value of the variable. You can not assign a new value to this tensor as it is not a

reference to the variable.

To avoid copies, if the consumer of the returned value is on the same device as the variable, this actually returns the live value of the variable, not a copy. Updates to the variable are seen by the consumer. If the consumer is on a different device it will get a copy of the variable.

Returns:

A `Tensor` containing the value of the variable.

Class Members

`__array_priority__`

Last updated November 2, 2017.