Tensor Transformations

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

Contents

  - `tf.squeeze(input, squeeze_dims=None, name=None)`
  - `tf.expand_dims(input, dim, name=None)`
  - `tf.meshgrid(*args, **kwargs)`
- Slicing and Joining
  - `tf.slice(input_, begin, size, name=None)`
  - `tf.strided_slice(input_, begin, end, strides, begin_mask=0, end_mask=0, ellipsis_mask=0, new_axis_mask=0, shrink_axis_mask=0, var=None, name=None)`
  - `tf.split(split_dim, num_split, value, name=split)`
  - `tf.tile(input, multiples, name=None)`
  - `tf.pad(tensor, paddings, mode=CONSTANT, name=None)`
  - `tf.concat(concat_dim, values, name=concat)`
  - `tf.pack(values, axis=0, name=pack)`
  - `tf.unpack(value, num=None, axis=0, name=unpack)`
  - `tf.reverse_sequence(input, seq_lengths, seq_dim, batch_dim=None, name=None)`
  - `tf.reverse(tensor, dims, name=None)`
  - `tf.transpose(a, perm=None, name=transpose)`
  - `tf.extract_image_patches(images, ksizes, strides, rates, padding, name=None)`
  - `tf.space_to_batch_nd(input, block_shape, paddings, name=None)`
  - `tf.space_to_batch(input, paddings, block_size, name=None)`
  - `tf.required_space_to_batch_paddings(input_shape, block_shape, base_paddings=None, name=None)`
  - `tf.batch_to_space_nd(input, block_shape, crops, name=None)`
  - `tf.batch_to_space(input, crops, block_size, name=None)`
  - `tf.space_to_depth(input, block_size, name=None)`

  - `tf.depth_to_space(input, block_size, name=None)`
  - `tf.gather(params, indices, validate_indices=None, name=None)`
  - `tf.gather_nd(params, indices, name=None)`
  - `tf.unique_with_counts(x, out_idx=None, name=None)`
  - `tf.dynamic_partition(data, partitions, num_partitions, name=None)`
  - `tf.dynamic_stitch(indices, data, name=None)`
  - `tf.boolean_mask(tensor, mask, name=boolean_mask)`
  - `tf.one_hot(indices, depth, on_value=None, off_value=None, axis=None, dtype=None, name=None)`
- Examples
  - `tf.sequence_mask(lengths, maxlen=None, dtype=tf.bool, name=None)`

Casting

TensorFlow provides several operations that you can use to cast tensor data types in your graph.

`tf.string_to_number(string_tensor, out_type=None, name=None)`

Converts each string in the input Tensor to the specified numeric type.

(Note that int32 overflow results in an error while float overflow results in a rounded value.)

Args:

- `string_tensor`: A `Tensor` of type `string`.
- `out_type`: An optional `tf.DType` from: `tf.float32, tf.int32`. Defaults to `tf.float32`. The numeric type to interpret each string in string_tensor as.

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

Returns:

A `Tensor` of type `out_type`. A Tensor of the same shape as the input `string_tensor`.

`tf.to_double(x, name='ToDouble')`

Casts a tensor to type `float64`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `float64`.

Raises:

* `TypeError`: If `x` cannot be cast to the `float64`.

`tf.to_float(x, name='ToFloat')`

Casts a tensor to type `float32`.

Args:

* `x`: A `Tensor` or `SparseTensor`.

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

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `float32`.

Raises:

* `TypeError`: If `x` cannot be cast to the `float32`.

`tf.to_bfloat16(x, name='ToBFloat16')`

Casts a tensor to type `bfloat16`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `bfloat16`.

Raises:

* `TypeError`: If `x` cannot be cast to the `bfloat16`.

`tf.to_int32(x, name='ToInt32')`

Casts a tensor to type `int32`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `int32`.

Raises:

* `TypeError`: If `x` cannot be cast to the `int32`.

`tf.to_int64(x, name='ToInt64')`

Casts a tensor to type `int64`.

Args:

* `x`: A `Tensor` or `SparseTensor`.
* `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x` with type `int64`.

Raises:

* `TypeError`: If `x` cannot be cast to the `int64`.

`tf.cast(x, dtype, name=None)`

Casts a tensor to a new type.

The operation casts `x` (in case of `Tensor`) or `x.values` (in case of `SparseTensor`) to `dtype`.

For example:

```
# tensor `a` is [1.8, 2.2], dtype=tf.float
tf.cast(a, tf.int32) ==> [1, 2]  # dtype=tf.int32
```

Args:

- `x`: A `Tensor` or `SparseTensor`.
- `dtype`: The destination type.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor` with same shape as `x`.

Raises:

- `TypeError`: If `x` cannot be cast to the `dtype`.

`tf.bitcast(input, type, name=None)`

Bitcasts a tensor from one type to another without copying data.

Given a tensor `input`, this operation returns a tensor that has the same buffer data as `input` with datatype `type`.

If the input datatype `T` is larger than the output datatype `type` then the shape changes from [...] to [..., sizeof(`T`)/sizeof(`type`)].

If `T` is smaller than `type`, the operator requires that the rightmost dimension be equal to sizeof(`type`)/sizeof(`T`). The shape then goes from [..., sizeof(`type`)/sizeof(`T`)] to [...].

NOTE: Bitcast is implemented as a low-level cast, so machines with different endian orderings will give different results.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- `type`: A `tf.DType` from: `tf.float32, tf.float64, tf.int64, tf.int32, tf.uint8, tf.uint16, tf.int16, tf.int8, tf.complex64, tf.complex128, tf.qint8, tf.quint8, tf.qint32, tf.half`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `type`.

`tf.saturate_cast(value, dtype, name=None)`

Performs a safe saturating cast of `value` to `dtype`.

This function casts the input to `dtype` without applying any scaling. If there is a danger that values would over or underflow in the cast, this op applies the appropriate clamping before the cast.

Args:

- `value`: A `Tensor`.
- `dtype`: The desired output `DType`.

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

Returns:

`value` safely cast to `dtype`.

Shapes and Shaping

TensorFlow provides several operations that you can use to determine the shape of a tensor and change the shape of a tensor.

`tf.shape(input, name=None, out_type=tf.int32)`

Returns the shape of a tensor.

This operation returns a 1-D integer tensor representing the shape of `input`.

For example:

```
# 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
shape(t) ==> [2, 2, 3]
```

Args:

- `input`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).
- `out_type`: (Optional) The specified output type of the operation (`int32` or `int64`). Defaults to tf.int32.

Returns:

A `Tensor` of type `out_type`.

`tf.shape_n(input, out_type=None, name=None)`

Returns shape of tensors.

This operation returns N 1-D integer tensors representing shape of `input[i]s`.

Args:

- `input`: A list of at least 1 `Tensor` objects of the same type.
- `out_type`: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int32`.
- `name`: A name for the operation (optional).

Returns:

A list with the same number of `Tensor` objects as `input` of `Tensor` objects of type out_type.

`tf.size(input, name=None, out_type=tf.int32)`

Returns the size of a tensor.

This operation returns an integer representing the number of elements in `input`.

For example:

```
# 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]]
size(t) ==> 12
```

Args:

- `input`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).
- `out_type`: (Optional) The specified output type of the operation (`int32` or `int64`). Defaults to tf.int32.

Returns:

A `Tensor` of type `out_type`. Defaults to tf.int32.

`tf.rank(input, name=None)`

Returns the rank of a tensor.

This operation returns an integer representing the rank of `input`.

For example:

```
# 't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
# shape of tensor 't' is [2, 2, 3]
rank(t) ==> 3
```

Note: The rank of a tensor is not the same as the rank of a matrix. The rank of a tensor is the number of indices required to uniquely select each element of the tensor. Rank is also known as "order", "degree", or "ndims."

Args:

- `input`: A `Tensor` or `SparseTensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

---

tf.reshape(tensor, shape, name=None)

Reshapes a tensor.

Given `tensor`, this operation returns a tensor that has the same values as `tensor` with shape `shape`.

If one component of `shape` is the special value -1, the size of that dimension is computed so that the total size remains constant. In particular, a `shape` of `[-1]` flattens into 1-D. At most one component of `shape` can be -1.

If `shape` is 1-D or higher, then the operation returns a tensor with shape `shape` filled with the values of `tensor`. In this case, the number of elements implied by `shape` must be the same as the number of elements in `tensor`.

For example:

```
# tensor 't' is [1, 2, 3, 4, 5, 6, 7, 8, 9]
# tensor 't' has shape [9]
reshape(t, [3, 3]) ==> [[1, 2, 3],
                        [4, 5, 6],
                        [7, 8, 9]]

# tensor 't' is [[[1, 1], [2, 2]],
#                [[3, 3], [4, 4]]]
# tensor 't' has shape [2, 2, 2]
reshape(t, [2, 4]) ==> [[1, 1, 2, 2],
                        [3, 3, 4, 4]]

# tensor 't' is [[[1, 1, 1],
#                 [2, 2, 2]],
#                [[3, 3, 3],
#                 [4, 4, 4]],
#                [[5, 5, 5],
#                 [6, 6, 6]]]
# tensor 't' has shape [3, 2, 3]
# pass '[-1]' to flatten 't'
```

```
reshape(t, [-1]) ==> [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6]

# -1 can also be used to infer the shape

# -1 is inferred to be 9:
reshape(t, [2, -1]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],
                         [4, 4, 4, 5, 5, 5, 6, 6, 6]]
# -1 is inferred to be 2:
reshape(t, [-1, 9]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],
                         [4, 4, 4, 5, 5, 5, 6, 6, 6]]
# -1 is inferred to be 3:
reshape(t, [ 2, -1, 3]) ==> [[[1, 1, 1],
                              [2, 2, 2],
                              [3, 3, 3]],
                             [[4, 4, 4],
                              [5, 5, 5],
                              [6, 6, 6]]]

# tensor 't' is [7]
# shape `[]` reshapes to a scalar
reshape(t, []) ==> 7
```

Args:

- `tensor`: A `Tensor`.
- `shape`: A `Tensor`. Must be one of the following types: `int32`, `int64`. Defines the shape of the output tensor.
- `name`: A name for the operation (optional).

Returns:

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

`tf.squeeze(input, squeeze_dims=None, name=None)`

Removes dimensions of size 1 from the shape of a tensor.

Given a tensor `input`, this operation returns a tensor of the same type with all dimensions of size 1 removed. If you don't want to remove all size 1 dimensions, you can remove specific size 1 dimensions by specifying `squeeze_dims`.

For example:

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t)) ==> [2, 3]
```

Or, to remove specific size 1 dimensions:

```
# 't' is a tensor of shape [1, 2, 1, 3, 1, 1]
shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

Args:

- `input`: A `Tensor`. The `input` to squeeze.
- `squeeze_dims`: An optional list of `ints`. Defaults to `[]`. If specified, only squeezes the dimensions listed. The dimension index starts at 0. It is an error to squeeze a dimension that is not 1.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Contains the same data as `input`, but has one or more dimensions of size 1 removed.

`tf.expand_dims(input, dim, name=None)`

Inserts a dimension of 1 into a tensor's shape.

Given a tensor `input`, this operation inserts a dimension of 1 at the dimension index `dim` of `input`'s shape. The dimension index `dim` starts at zero; if you specify a negative number for `dim` it is counted backward from the end.

This operation is useful if you want to add a batch dimension to a single element. For example, if you have a single image of shape `[height, width, channels]`, you can make it a batch of 1 image with `expand_dims(image, 0)`, which will make the shape `[1, height, width, channels]`.

Other examples:

```
# 't' is a tensor of shape [2]
shape(expand_dims(t, 0)) ==> [1, 2]
shape(expand_dims(t, 1)) ==> [2, 1]
shape(expand_dims(t, -1)) ==> [2, 1]

# 't2' is a tensor of shape [2, 3, 5]
shape(expand_dims(t2, 0)) ==> [1, 2, 3, 5]
shape(expand_dims(t2, 2)) ==> [2, 3, 1, 5]
shape(expand_dims(t2, 3)) ==> [2, 3, 5, 1]
```

This operation requires that:

`-1-input.dims() <= dim <= input.dims()`

This operation is related to `squeeze()`, which removes dimensions of size 1.

Args:

- `input`: A `Tensor`.
- `dim`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 0-D (scalar). Specifies the dimension index at which to expand the shape of `input`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. Contains the same data as `input`, but its shape has an additional dimension of size 1 added.

`tf.meshgrid(*args, **kwargs)`

Broadcasts parameters for evaluation on an N-D grid.

Given N one-dimensional coordinate arrays `*args`, returns a list `outputs` of N-D coordinate arrays for evaluating expressions on an N-D grid.

Notes:

`meshgrid` supports cartesian ('xy') and matrix ('ij') indexing conventions. When the `indexing` argument is set to 'xy' (the default), the broadcasting instructions for the first two dimensions are swapped.

Examples:

Calling `X, Y = meshgrid(x, y)` with the tensors

```
x = [1, 2, 3]
y = [4, 5, 6]
```

results in

```
X = [[1, 1, 1],
     [2, 2, 2],
     [3, 3, 3]]
Y = [[4, 5, 6],
     [4, 5, 6],
     [4, 5, 6]]
```

Args:

- `*args`: `Tensor`s with rank 1
- `indexing`: Either 'xy' or 'ij' (optional, default: 'xy')
- `name`: A name for the operation (optional).

Returns:

- `outputs`: A list of N `Tensor`s with rank N

Slicing and Joining

TensorFlow provides several operations to slice or extract parts of a tensor, or join multiple tensors together.

`tf.slice(input_, begin, size, name=None)`

Extracts a slice from a tensor.

This operation extracts a slice of size `size` from a tensor `input` starting at the location specified by `begin`. The slice `size` is represented as a tensor shape, where `size[i]` is the number of elements of the 'i'th dimension of `input` that you want to slice. The starting location (`begin`) for the slice is represented as an offset in each dimension of `input`. In other words, `begin[i]` is the offset into the 'i'th dimension of `input` that you want to slice from.

`begin` is zero-based; `size` is one-based. If `size[i]` is -1, all remaining elements in dimension i are included in the slice. In other words, this is equivalent to setting:

`size[i] = input.dim_size(i) - begin[i]`

This operation requires that:

`0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n]`

For example:

```
# 'input' is [[[1, 1, 1], [2, 2, 2]],
#             [[3, 3, 3], [4, 4, 4]],
#             [[5, 5, 5], [6, 6, 6]]]
tf.slice(input, [1, 0, 0], [1, 1, 3]) ==> [[[3, 3, 3]]]
tf.slice(input, [1, 0, 0], [1, 2, 3]) ==> [[[3, 3, 3],
                                            [4, 4, 4]]]
```

```
tf.slice(input, [1, 0, 0], [2, 1, 3]) ==> [[[3, 3, 3]],
                                           [[5, 5, 5]]]
```

Args:

- `input_`: A `Tensor`.
- `begin`: An `int32` or `int64` `Tensor`.
- `size`: An `int32` or `int64` `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` the same type as `input`.

---

tf.strided_slice(input_, begin, end, strides, begin_mask=0, end_mask=0, ellipsis_mask=0, new_axis_mask=0, shrink_axis_mask=0, var=None, name=None)

Extracts a strided slice from a tensor.

To a first order, this operation extracts a slice of size `end - begin` from a tensor `input` starting at the location specified by `begin`. The slice continues by adding `stride` to the `begin` index until all dimensions are not less than `end`. Note that components of stride can be negative, which causes a reverse slice.

This operation can be thought of an encoding of a numpy style sliced range. Given a python slice input[, , ...,] this function will be called as follows.

`begin`, `end`, and `strides` will be all length n. n is in general not the same dimensionality as `input`.

For the ith spec, `begin_mask`, `end_mask`, `ellipsis_mask`, `new_axis_mask`, and `shrink_axis_mask` will have the ith bit corresponding to the ith spec.

If the ith bit of `begin_mask` is non-zero, `begin[i]` is ignored and the fullest possible range in that dimension is used instead. `end_mask` works analogously, except with the end range.

`foo[5:,:,:3]` on a 7x8x9 tensor is equivalent to `foo[5:7,0:8,0:3]`. `foo[::-1]` reverses a tensor with shape 8.

If the ith bit of `ellipsis_mask`, as many unspecified dimensions as needed will be inserted between other dimensions. Only one non-zero bit is allowed in `ellipsis_mask`.

For example `foo[3:5,...,4:5]` on a shape 10x3x3x10 tensor is equivalent to `foo[3:5,:,:,4:5]` and `foo[3:5,...]` is equivalent to `foo[3:5,:,:,:]`.

If the ith bit of `new_axis_mask` is one, then a `begin`, `end`, and `stride` are ignored and a new length 1 dimension is added at this point in the output tensor.

For example `foo[3:5,4]` on a 10x8 tensor produces a shape 2 tensor whereas `foo[3:5,4:5]` produces a shape 2x1 tensor with shrink_mask being 1<<1 == 2.

If the ith bit of `shrink_axis_mask` is one, then `begin`, `end[i]`, and `stride[i]` are used to do a slice in the appropriate dimension, but the output tensor will be reduced in dimensionality by one. This is only valid if the ith entry of slice[i]==1.

NOTE: `begin` and `end` are zero-indexed`.`strides` entries must be non-zero.

```
# 'input' is [[[1, 1, 1], [2, 2, 2]],
#             [[3, 3, 3], [4, 4, 4]],
#             [[5, 5, 5], [6, 6, 6]]]
tf.slice(input, [1, 0, 0], [2, 1, 3], [1, 1, 1]) ==> [[[3, 3, 3]]]
tf.slice(input, [1, 0, 0], [2, 2, 3], [1, 1, 1]) ==> [[[3, 3, 3],
                                                       [4, 4, 4]]]
tf.slice(input, [1, 1, 0], [2, -1, 3], [1, -1, 1]) ==>[[[4, 4, 4],
                                                        [3, 3, 3]]]
```

Args:

- `input_`: A `Tensor`.

* `begin`: An `int32` or `int64` `Tensor`.
* `end`: An `int32` or `int64` `Tensor`.
* `strides`: An `int32` or `int64` `Tensor`.
* `begin_mask`: An `int32` mask.
* `end_mask`: An `int32` mask.
* `ellipsis_mask`: An `int32` mask.
* `new_axis_mask`: An `int32` mask.
* `shrink_axis_mask`: An `int32` mask.
* `var`: The variable coresponding to `input_` or None
* `name`: A name for the operation (optional).

Returns:

A `Tensor` the same type as `input`.

`tf.split(split_dim, num_split, value, name='split')`

Splits a tensor into `num_split` tensors along one dimension.

Splits `value` along dimension `split_dim` into `num_split` smaller tensors. Requires that `num_split` evenly divide `value.shape[split_dim]`.

For example:

```
# 'value' is a tensor with shape [5, 30]
# Split 'value' into 3 tensors along dimension 1
split0, split1, split2 = tf.split(1, 3, value)
tf.shape(split0) ==> [5, 10]
```

Note: If you are splitting along an axis by the length of that axis, consider using unpack, e.g.

```
num_items = t.get_shape()[axis].value
[tf.squeeze(s, [axis]) for s in tf.split(axis, num_items, t)]
```

can be rewritten as

```
tf.unpack(t, axis=axis)
```

Args:

- `split_dim`: A 0-D `int32` `Tensor`. The dimension along which to split. Must be in the range `[0, rank(value))`.
- `num_split`: A Python integer. The number of ways to split.
- `value`: The `Tensor` to split.
- `name`: A name for the operation (optional).

Returns:

`num_split` `Tensor` objects resulting from splitting `value`.

`tf.tile(input, multiples, name=None)`

Constructs a tensor by tiling a given tensor.

This operation creates a new tensor by replicating `input` `multiples` times. The output tensor's i'th dimension has `input.dims(i) * multiples[i]` elements, and the values of `input` are replicated `multiples[i]` times along the 'i'th dimension. For example, tiling `[a b c d]` by `[2]` produces `[a b c d a b c d]`.

Args:

- `input`: A `Tensor`. 1-D or higher.
- `multiples`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 1-D. Length must be the same as the number of dimensions in `input`
- `name`: A name for the operation (optional).

Returns:

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

`tf.pad(tensor, paddings, mode='CONSTANT', name=None)`

Pads a tensor.

This operation pads a `tensor` according to the `paddings` you specify. `paddings` is an integer tensor with shape `[n, 2]`, where n is the rank of `tensor`. For each dimension D of `input`, `paddings[D, 0]` indicates how many values to add before the contents of `tensor` in that dimension, and `paddings[D, 1]` indicates how many values to add after the contents of `tensor` in that dimension. If `mode` is "REFLECT" then both `paddings[D, 0]` and `paddings[D, 1]` must be no greater than `tensor.dim_size(D) - 1`. If `mode` is "SYMMETRIC" then both `paddings[D, 0]` and `paddings[D, 1]` must be no greater than `tensor.dim_size(D)`.

The padded size of each dimension D of the output is:

`paddings[D, 0] + tensor.dim_size(D) + paddings[D, 1]`

For example:

```
# 't' is [[1, 2, 3], [4, 5, 6]].
# 'paddings' is [[1, 1,], [2, 2]].
# rank of 't' is 2.
pad(t, paddings, "CONSTANT") ==> [[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]]

pad(t, paddings, "REFLECT") ==> [[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]]

pad(t, paddings, "SYMMETRIC") ==> [[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]]
```

Args:

- `tensor`: A `Tensor`.
- `paddings`: A `Tensor` of type `int32`.
- `mode`: One of "CONSTANT", "REFLECT", or "SYMMETRIC".
- `name`: A name for the operation (optional).

Returns:

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

Raises:

- `ValueError`: When mode is not one of "CONSTANT", "REFLECT", or "SYMMETRIC".

`tf.concat(concat_dim, values, name='concat')`

Concatenates tensors along one dimension.

Concatenates the list of tensors `values` along dimension `concat_dim`. If `values[i].shape = [D0, D1, ... Dconcat_dim(i), ...Dn]`, the concatenated result has shape

```
[D0, D1, ... Rconcat_dim, ...Dn]
```

where

```
Rconcat_dim = sum(Dconcat_dim(i))
```

That is, the data from the input tensors is joined along the `concat_dim` dimension.

The number of dimensions of the input tensors must match, and all dimensions except `concat_dim` must be equal.

For example:

```
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
tf.concat(0, [t1, t2]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]]

# tensor t3 with shape [2, 3]
# tensor t4 with shape [2, 3]
tf.shape(tf.concat(0, [t3, t4])) ==> [4, 3]
tf.shape(tf.concat(1, [t3, t4])) ==> [2, 6]
```

Note: If you are concatenating along a new axis consider using pack. E.g.

```
tf.concat(axis, [tf.expand_dims(t, axis) for t in tensors])
```

can be rewritten as

```
tf.pack(tensors, axis=axis)
```

Args:

- `concat_dim`: 0-D `int32` `Tensor`. Dimension along which to concatenate.
- `values`: A list of `Tensor` objects or a single `Tensor`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor` resulting from concatenation of the input tensors.

`tf.pack(values, axis=0, name='pack')`

Packs a list of rank-`R` tensors into one rank-`(R+1)` tensor.

Packs the list of tensors in `values` into a tensor with rank one higher than each tensor in `values`, by packing them along the `axis` dimension. Given a list of length `N` of tensors of shape `(A, B, C)`;

if `axis == 0` then the `output` tensor will have the shape `(N, A, B, C)`. if `axis == 1` then the `output` tensor will have the shape `(A, N, B, C)`. Etc.

For example:

```
# 'x' is [1, 4]
# 'y' is [2, 5]
# 'z' is [3, 6]
pack([x, y, z]) => [[1, 4], [2, 5], [3, 6]]  # Pack along first dim.
pack([x, y, z], axis=1) => [[1, 2, 3], [4, 5, 6]]
```

This is the opposite of unpack. The numpy equivalent is

```
tf.pack([x, y, z]) = np.asarray([x, y, z])
```

Args:

- `values`: A list of `Tensor` objects with the same shape and type.
- `axis`: An `int`. The axis to pack along. Defaults to the first dimension. Supports negative indexes.
- `name`: A name for this operation (optional).

Returns:

- `output`: A packed `Tensor` with the same type as `values`.

Raises:

- `ValueError`: If `axis` is out of the range [-(R+1), R+1).

`tf.unpack(value, num=None, axis=0, name='unpack')`

Unpacks the given dimension of a rank-`R` tensor into rank-`(R-1)` tensors.

Unpacks `num` tensors from `value` by chipping it along the `axis` dimension. If `num` is not specified (the default), it is inferred from `value`'s shape. If `value.shape[axis]` is not known, `ValueError` is raised.

For example, given a tensor of shape `(A, B, C, D)`;

If `axis == 0` then the i'th tensor in `output` is the slice `value[i, :, :, :]` and each tensor in `output` will have shape `(B, C, D)`. (Note that the dimension unpacked along is gone, unlike `split`).

If `axis == 1` then the i'th tensor in `output` is the slice `value[:, i, :, :]` and each tensor in `output` will have shape `(A, C, D)`. Etc.

This is the opposite of pack. The numpy equivalent is

```
tf.unpack(x, n) = list(x)
```

Args:

* `value`: A rank `R > 0` `Tensor` to be unpacked.
* `num`: An `int`. The length of the dimension `axis`. Automatically inferred if `None` (the default).
* `axis`: An `int`. The axis to unpack along. Defaults to the first dimension. Supports negative indexes.
* `name`: A name for the operation (optional).

Returns:

The list of `Tensor` objects unpacked from `value`.

Raises:

* `ValueError`: If `num` is unspecified and cannot be inferred.
* `ValueError`: If `axis` is out of the range [-R, R).

`tf.reverse_sequence(input, seq_lengths, seq_dim, batch_dim=None, name=None)`

Reverses variable length slices.

This op first slices `input` along the dimension `batch_dim`, and for each slice `i`, reverses the first `seq_lengths[i]` elements along the dimension `seq_dim`.

The elements of `seq_lengths` must obey `seq_lengths[i] < input.dims[seq_dim]`, and `seq_lengths` must be a vector of length `input.dims[batch_dim]`.

The output slice `i` along dimension `batch_dim` is then given by input slice `i`, with the first `seq_lengths[i]` slices along dimension `seq_dim` reversed.

For example:

```
# Given this:
batch_dim = 0
seq_dim = 1
input.dims = (4, 8, ...)
seq_lengths = [7, 2, 3, 5]

# then slices of input are reversed on seq_dim, but only up to seq_lengths:
output[0, 0:7, :, ...] = input[0, 7:0:-1, :, ...]
output[1, 0:2, :, ...] = input[1, 2:0:-1, :, ...]
output[2, 0:3, :, ...] = input[2, 3:0:-1, :, ...]
output[3, 0:5, :, ...] = input[3, 5:0:-1, :, ...]

# while entries past seq_lens are copied through:
output[0, 7:, :, ...] = input[0, 7:, :, ...]
output[1, 2:, :, ...] = input[1, 2:, :, ...]
output[2, 3:, :, ...] = input[2, 3:, :, ...]
output[3, 2:, :, ...] = input[3, 2:, :, ...]
```

In contrast, if:

```
# Given this:
batch_dim = 2
seq_dim = 0
input.dims = (8, ?, 4, ...)
seq_lengths = [7, 2, 3, 5]

# then slices of input are reversed on seq_dim, but only up to seq_lengths:
output[0:7, :, 0, :, ...] = input[7:0:-1, :, 0, :, ...]
output[0:2, :, 1, :, ...] = input[2:0:-1, :, 1, :, ...]
output[0:3, :, 2, :, ...] = input[3:0:-1, :, 2, :, ...]
output[0:5, :, 3, :, ...] = input[5:0:-1, :, 3, :, ...]

# while entries past seq_lens are copied through:
output[7:, :, 0, :, ...] = input[7:, :, 0, :, ...]
output[2:, :, 1, :, ...] = input[2:, :, 1, :, ...]
output[3:, :, 2, :, ...] = input[3:, :, 2, :, ...]
output[2:, :, 3, :, ...] = input[2:, :, 3, :, ...]
```

Args:

- `input`: A `Tensor`. The input to reverse.

- `seq_lengths`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 1-D with length `input.dims(batch_dim)` and `max(seq_lengths) < input.dims(seq_dim)`
- `seq_dim`: An `int`. The dimension which is partially reversed.
- `batch_dim`: An optional `int`. Defaults to `0`. The dimension along which reversal is performed.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. The partially reversed input. It has the same shape as `input`.

`tf.reverse(tensor, dims, name=None)`

Reverses specific dimensions of a tensor.

Given a `tensor`, and a `bool` tensor `dims` representing the dimensions of `tensor`, this operation reverses each dimension i of `tensor` where `dims[i]` is `True`.

`tensor` can have up to 8 dimensions. The number of dimensions of `tensor` must equal the number of elements in `dims`. In other words:

`rank(tensor) = size(dims)`

For example:

```
# tensor 't' is [[[[ 0,  1,  2,  3],
#                  [ 4,  5,  6,  7],
#                  [ 8,  9, 10, 11]],
#                 [[12, 13, 14, 15],
#                  [16, 17, 18, 19],
#                  [20, 21, 22, 23]]]]
# tensor 't' shape is [1, 2, 3, 4]

# 'dims' is [False, False, False, True]
reverse(t, dims) ==> [[[[ 3,  2,  1,  0],
                        [ 7,  6,  5,  4],
                        [ 11, 10, 9, 8]],
```

```
                          [[15, 14, 13, 12],
                           [19, 18, 17, 16],
                           [23, 22, 21, 20]]]]

# 'dims' is [False, True, False, False]
reverse(t, dims) ==> [[[[12, 13, 14, 15],
                         [16, 17, 18, 19],
                         [20, 21, 22, 23]
                        [[ 0,  1,  2,  3],
                         [ 4,  5,  6,  7],
                         [ 8,  9, 10, 11]]]]

# 'dims' is [False, False, True, False]
reverse(t, dims) ==> [[[[8, 9, 10, 11],
                         [4, 5, 6, 7],
                         [0, 1, 2, 3]]
                        [[20, 21, 22, 23],
                         [16, 17, 18, 19],
                         [12, 13, 14, 15]]]]
```

Args:

- `tensor`: A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int32`, `int64`, `bool`, `half`, `float32`, `float64`, `complex64`, `complex128`. Up to 8-D.
- `dims`: A `Tensor` of type `bool`. 1-D. The dimensions to reverse.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `tensor`. The same shape as `tensor`.

`tf.transpose(a, perm=None, name='transpose')`

Transposes `a`. Permutes the dimensions according to `perm`.

The returned tensor's dimension i will correspond to the input dimension `perm[i]`. If `perm` is not given, it is set to (n-1...0), where n is the rank of the input tensor. Hence by default, this operation performs a regular matrix transpose on 2-D input Tensors.

For example:

```
# 'x' is [[1 2 3]
#         [4 5 6]]
tf.transpose(x) ==> [[1 4]
                     [2 5]
                     [3 6]]

# Equivalently
tf.transpose(x, perm=[1, 0]) ==> [[1 4]
                                  [2 5]
                                  [3 6]]

# 'perm' is more useful for n-dimensional tensors, for n > 2
# 'x' is   [[[1  2  3]
#            [4  5  6]]
#           [[7  8  9]
#            [10 11 12]]]
# Take the transpose of the matrices in dimension-0
tf.transpose(x, perm=[0, 2, 1]) ==> [[[1  4]
                                      [2  5]
                                      [3  6]]

                                     [[7 10]
                                      [8 11]
                                      [9 12]]]
```

Args:

- `a`: A `Tensor`.
- `perm`: A permutation of the dimensions of `a`.
- `name`: A name for the operation (optional).

Returns:

A transposed `Tensor`.

tf.extract_image_patches(images, ksizes, strides, rates, padding, name=None)

Extract `patches` from `images` and put them in the "depth" output dimension.

Args:

- `images`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`. 4-D Tensor with shape `[batch, in_rows, in_cols, depth]`.
- `ksizes`: A list of `ints` that has length `>= 4`. The size of the sliding window for each dimension of `images`.
- `strides`: A list of `ints` that has length `>= 4`. 1-D of length 4. How far the centers of two consecutive patches are in the images. Must be: `[1, stride_rows, stride_cols, 1]`.
- `rates`: A list of `ints` that has length `>= 4`. 1-D of length 4. Must be: `[1, rate_rows, rate_cols, 1]`. This is the input stride, specifying how far two consecutive patch samples are in the input. Equivalent to extracting patches with `patch_sizes_eff = patch_sizes + (patch_sizes - 1) * (rates - 1), followed by subsampling them spatially by a factor of`rates`.

  `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.

  We specify the size-related attributes as:

  ```
  ksizes = [1, ksize_rows, ksize_cols, 1]
  strides = [1, strides_rows, strides_cols, 1]
  rates = [1, rates_rows, rates_cols, 1]
  ```

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

Returns:

A `Tensor`. Has the same type as `images`. 4-D Tensor with shape `[batch, out_rows, out_cols, ksize_rows * ksize_cols * depth]` containing image patches with size `ksize_rows x ksize_cols x depth` vectorized in the "depth" dimension.

tf.space_to_batch_nd(input, block_shape, paddings, name=None)

SpaceToBatch for N-D tensors of type T.

This operation divides "spatial" dimensions `[1, ..., M]` of the input into a grid of blocks of shape `block_shape`, and interleaves these blocks with the "batch" dimension (0) such that in the output, the spatial dimensions `[1, ..., M]` correspond to the position within the grid, and the batch dimension combines both the position within a spatial block and the original batch position. Prior to division into blocks, the spatial dimensions of the input are optionally zero padded according to `paddings`. See below for a precise description.

Args:

- `input`: A `Tensor`. N-D with shape `input_shape = [batch] + spatial_shape + remaining_shape`, where spatial_shape has `M` dimensions.

- `block_shape`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 1-D with shape `[M]`, all values must be >= 1.

  `paddings`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 2-D with shape `[M, 2]`, all values must be >= 0. `paddings[i] = [pad_start, pad_end]` specifies the padding for input dimension `i + 1`, which corresponds to spatial dimension `i`. It is required that `block_shape[i]` divides `input_shape[i + 1] + pad_start + pad_end`.

  This operation is equivalent to the following steps:

    Zero-pad the start and end of dimensions `[1, ..., M]` of the input according to `paddings` to produce `padded` of shape `padded_shape`.

    Reshape `padded` to `reshaped_padded` of shape: [batch] + [padded_shape[1] / block_shape[0], block_shape[0], ..., padded_shape[M] / block_shape[M-1], block_shape[M-1]] + remaining_shape

    Permute dimensions of `reshaped_padded` to produce `permuted_reshaped_padded` of shape: block_shape + [batch] + [padded_shape[1] / block_shape[0], ..., padded_shape[M] /

block_shape[M-1]] + remaining_shape

Reshape `permuted_reshaped_padded` to flatten `block_shape` into the batch dimension, producing an output tensor of shape: [batch * prod(block_shape)] + [padded_shape[1] / block_shape[0], ..., padded_shape[M] / block_shape[M-1]] + remaining_shape

Some examples:

(1) For the following input of shape `[1, 2, 2, 1]`, `block_shape = [2, 2]`, and `paddings = [[0, 0], [0, 0]]`:

```
x = [[[[1], [2]], [[3], [4]]]]
```

The output tensor has shape `[4, 1, 1, 1]` and value:

```
[[[[1]]], [[[2]]], [[[3]]], [[[4]]]]
```

(2) For the following input of shape `[1, 2, 2, 3]`, `block_shape = [2, 2]`, and `paddings = [[0, 0], [0, 0]]`:

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

The output tensor has shape `[4, 1, 1, 3]` and value:

```
[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]], [[10, 11, 12]]]
```

(3) For the following input of shape `[1, 4, 4, 1]`, `block_shape = [2, 2]`, and `paddings = [[0, 0], [0, 0]]`:

```
x = [[[[1],   [2],  [3],  [4]],
      [[5],   [6],  [7],  [8]],
      [[9],  [10], [11],  [12]],
      [[13], [14], [15],  [16]]]]
```

The output tensor has shape `[4, 2, 2, 1]` and value:

```
x = [[[[1], [3]], [[5], [7]]],
     [[[2], [4]], [[10], [12]]],
     [[[5], [7]], [[13], [15]]],
     [[[6], [8]], [[14], [16]]]]
```

(4) For the following input of shape `[2, 2, 4, 1]`, block_shape = `[2, 2]`, and paddings = `[[0, 0], [2, 0]]`:

```
x = [[[[1],   [2],  [3],  [4]],
      [[5],   [6],  [7],  [8]]],
     [[[9],  [10], [11],  [12]],
      [[13], [14], [15],  [16]]]]
```

The output tensor has shape `[8, 1, 3, 1]` and value:

```
x = [[[[0], [1], [3]]], [[[0], [9], [11]]],
     [[[0], [2], [4]]], [[[0], [10], [12]]],
     [[[0], [5], [7]]], [[[0], [13], [15]]],
     [[[0], [6], [8]]], [[[0], [14], [16]]]]
```

Among others, this operation is useful for reducing atrous convolution into regular convolution.

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

Returns:

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

tf.space_to_batch(input, paddings, block_size, name=None)

SpaceToBatch for 4-D tensors of type T.

This is a legacy version of the more general SpaceToBatchND.

Zero-pads and then rearranges (permutes) blocks of spatial data into batch. More specifically, this op outputs a copy of the input tensor where values from the `height` and `width` dimensions are moved to the `batch` dimension. After the zero-padding, both `height` and `width` of the input must be divisible by the block size.

Args:

- `input`: A `Tensor`. 4-D with shape `[batch, height, width, depth]`.

  `paddings`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 2-D tensor of non-negative integers with shape `[2, 2]`. It specifies the padding of the input with zeros across the spatial dimensions as follows:

  ```
  paddings = [[pad_top, pad_bottom], [pad_left, pad_right]]
  ```

  The effective spatial dimensions of the zero-padded input tensor will be:

  ```
  height_pad = pad_top + height + pad_bottom
  width_pad = pad_left + width + pad_right
  ```

  The attr `block_size` must be greater than one. It indicates the block size.

  - Non-overlapping blocks of size `block_size x block size` in the height and width dimensions are rearranged into the batch dimension at each location.
  - The batch of the output tensor is `batch * block_size * block_size`.
  - Both height_pad and width_pad must be divisible by block_size.

The shape of the output will be:

```
[batch*block_size*block_size, height_pad/block_size, width_pad/block_size,
 depth]
```

Some examples:

(1) For the following input of shape `[1, 2, 2, 1]` and block_size of 2:

```
x = [[[[1], [2]], [[3], [4]]]]
```

The output tensor has shape `[4, 1, 1, 1]` and value:

```
[[[[1]]], [[[2]]], [[[3]]], [[[4]]]]
```

(2) For the following input of shape `[1, 2, 2, 3]` and block_size of 2:

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

The output tensor has shape `[4, 1, 1, 3]` and value:

```
[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]], [[10, 11, 12]]]
```

(3) For the following input of shape `[1, 4, 4, 1]` and block_size of 2:

```
x = [[[[1],   [2],  [3],  [4]],
      [[5],   [6],  [7],  [8]],
      [[9],  [10], [11],  [12]],
      [[13], [14], [15],  [16]]]]
```

The output tensor has shape `[4, 2, 2, 1]` and value:

```
x = [[[[1], [3]], [[5], [7]]],
     [[[2], [4]], [[10], [12]]],
     [[[5], [7]], [[13], [15]]],
     [[[6], [8]], [[14], [16]]]]
```

(4) For the following input of shape `[2, 2, 4, 1]` and block_size of 2:

```
x = [[[[1],   [2],  [3],  [4]],
       [[5],   [6],  [7],  [8]]],
     [[[9],  [10], [11],  [12]],
       [[13], [14], [15],  [16]]]]
```

The output tensor has shape `[8, 1, 2, 1]` and value:

```
x = [[[[1], [3]]], [[[9], [11]]], [[[2], [4]]], [[[10], [12]]],
     [[[5], [7]]], [[[13], [15]]], [[[6], [8]]], [[[14], [16]]]]
```

Among others, this operation is useful for reducing atrous convolution into regular convolution.

`block_size`: An `int` that is `>= 2`.

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

Returns:

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

tf.required_space_to_batch_paddings(input_shape, block_shape, base_paddings=None, name=None)

Calculate padding required to make block_shape divide input_shape.

This function can be used to calculate a suitable paddings argument for use with space_to_batch_nd and batch_to_space_nd.

Args:

- `input_shape`: int32 Tensor of shape [N].
- `block_shape`: int32 Tensor of shape [N].
- `base_paddings`: Optional int32 Tensor of shape [N, 2]. Specifies the minimum amount of padding to use. All elements must be >= 0. If not specified, defaults to 0.
- `name`: string. Optional name prefix.

Returns:

(paddings, crops), where:

`paddings` and `crops` are int32 Tensors of rank 2 and shape [N, 2]

> `satisfying`:
>
> paddings[i, 0] = base_paddings[i, 0]. 0 <= paddings[i, 1] - base_paddings[i, 1] < block_shapei % block_shape[i] == 0
>
> crops[i, 0] = 0 crops[i, 1] = paddings[i, 1] - base_paddings[i, 1]
>
> `Raises`: ValueError if called with incompatible shapes.

---

tf.batch_to_space_nd(input, block_shape, crops, name=None)

BatchToSpace for N-D tensors of type T.

This operation reshapes the "batch" dimension 0 into `M + 1` dimensions of shape `block_shape + [batch]`, interleaves these blocks back into the grid defined by the spatial dimensions `[1, ..., M]`, to obtain a result with the same rank as the input. The spatial dimensions of this intermediate result are then optionally cropped according to `crops` to produce the output. This is the reverse of SpaceToBatch. See below for a precise description.

Args:

- `input`: A `Tensor`. N-D with shape `input_shape = [batch] + spatial_shape + remaining_shape`, where spatial_shape has M dimensions.
- `block_shape`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 1-D with shape `[M]`, all values must be >= 1.

  `crops`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 2-D with shape `[M, 2]`, all values must be >= 0. `crops[i] = [crop_start, crop_end]` specifies the amount to crop from input dimension `i + 1`, which corresponds to spatial dimension `i`. It is required that `crop_start[i] + crop_end[i] <= block_shape[i] * input_shape[i + 1]`.

  This operation is equivalent to the following steps:

  > Reshape `input` to `reshaped` of shape: [block_shape[0], ..., block_shape[M-1], batch / prod(block_shape), input_shape[1], ..., input_shape[N-1]]
  >
  > Permute dimensions of `reshaped` to produce `permuted` of shape [batch / prod(block_shape),
  >
  > input_shape[1], block_shape[0], ..., input_shape[M], block_shape[M-1],
  >
  > input_shape[M+1], ..., input_shape[N-1]]
  >
  > Reshape `permuted` to produce `reshaped_permuted` of shape [batch / prod(block_shape),
  >
  > input_shape[1] * block_shape[0], ..., input_shape[M] * block_shape[M-1],
  >
  > input_shape[M+1], ..., input_shape[N-1]]

Crop the start and end of dimensions `[1, ..., M]` of `reshaped_permuted` according to `crops` to produce the output of shape: [batch / prod(block_shape),

input_shape[1] * block_shape[0] - crops[0,0] - crops[0,1], ..., input_shape[M] * block_shape[M-1] - crops[M-1,0] - crops[M-1,1],

input_shape[M+1], ..., input_shape[N-1]]

Some examples:

(1) For the following input of shape `[4, 1, 1, 1]`, `block_shape = [2, 2]`, and `crops = [[0, 0], [0, 0]]`:

```
[[[[1]]], [[[2]]], [[[3]]], [[[4]]]]
```

The output tensor has shape `[1, 2, 2, 1]` and value:

```
x = [[[[1], [2]], [[3], [4]]]]
```

(2) For the following input of shape `[4, 1, 1, 3]`, `block_shape = [2, 2]`, and `crops = [[0, 0], [0, 0]]`:

```
[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]], [[10, 11, 12]]]
```

The output tensor has shape `[1, 2, 2, 3]` and value:

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

(3) For the following input of shape `[4, 2, 2, 1]`, `block_shape = [2, 2]`, and `crops = [[0, 0], [0, 0]]`:

```
x = [[[[1], [3]], [[5], [7]]],
     [[[2], [4]], [[10], [12]]],
     [[[5], [7]], [[13], [15]]],
     [[[6], [8]], [[14], [16]]]]
```

The output tensor has shape `[1, 4, 4, 1]` and value:

```
x = [[[1],   [2],  [3],  [4]],
     [[5],   [6],  [7],  [8]],
     [[9],  [10], [11],  [12]],
     [[13], [14], [15],  [16]]]
```

(4) For the following input of shape `[8, 1, 3, 1]`, `block_shape = [2, 2]`, and `crops = [[0, 0], [2, 0]]`:

```
x = [[[[0], [1], [3]]], [[[0], [9], [11]]],
     [[[0], [2], [4]]], [[[0], [10], [12]]],
     [[[0], [5], [7]]], [[[0], [13], [15]]],
     [[[0], [6], [8]]], [[[0], [14], [16]]]]
```

The output tensor has shape `[2, 2, 4, 1]` and value:

```
x = [[[[1],   [2],  [3],  [4]],
      [[5],   [6],  [7],  [8]]],
     [[[9],  [10], [11],  [12]],
      [[13], [14], [15],  [16]]]]
```

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

Returns:

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

`tf.batch_to_space(input, crops, block_size, name=None)`

BatchToSpace for 4-D tensors of type T.

This is a legacy version of the more general BatchToSpaceND.

Rearranges (permutes) data from batch into blocks of spatial data, followed by cropping. This is the reverse transformation of SpaceToBatch. More specifically, this op outputs a copy of the input tensor where values from the `batch` dimension are moved in spatial blocks to the `height` and `width` dimensions, followed by cropping along the `height` and `width` dimensions.

Args:

- `input`: A `Tensor`. 4-D tensor with shape `[batch*block_size*block_size, height_pad/block_size, width_pad/block_size, depth]`. Note that the batch size of the input tensor must be divisible by `block_size * block_size`.

  `crops`: A `Tensor`. Must be one of the following types: `int32`, `int64`. 2-D tensor of non-negative integers with shape `[2, 2]`. It specifies how many elements to crop from the intermediate result across the spatial dimensions as follows:

  ```
  crops = [[crop_top, crop_bottom], [crop_left, crop_right]]
  ```

  `block_size`: An `int` that is `>= 2`.

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

Returns:

A `Tensor`. Has the same type as `input`. 4-D with shape `[batch, height, width, depth]`, where:

```
height = height_pad - crop_top - crop_bottom
width = width_pad - crop_left - crop_right
```

The attr `block_size` must be greater than one. It indicates the block size.

Some examples:

(1) For the following input of shape `[4, 1, 1, 1]` and block_size of 2:

```
[[[[1]]], [[[2]]], [[[3]]], [[[4]]]]
```

The output tensor has shape `[1, 2, 2, 1]` and value:

```
x = [[[[1], [2]], [[3], [4]]]]
```

(2) For the following input of shape `[4, 1, 1, 3]` and block_size of 2:

```
[[[1, 2, 3]], [[4, 5, 6]], [[7, 8, 9]], [[10, 11, 12]]]
```

The output tensor has shape `[1, 2, 2, 3]` and value:

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

(3) For the following input of shape `[4, 2, 2, 1]` and block_size of 2:

```
x = [[[[1], [3]], [[5], [7]]],
     [[[2], [4]], [[10], [12]]],
     [[[5], [7]], [[13], [15]]],
     [[[6], [8]], [[14], [16]]]]
```

The output tensor has shape `[1, 4, 4, 1]` and value:

```
x = [[[1],   [2],  [3],  [4]],
     [[5],   [6],  [7],  [8]],
     [[9],  [10], [11],  [12]],
     [[13], [14], [15],  [16]]]
```

(4) For the following input of shape `[8, 1, 2, 1]` and block_size of 2:

```
x = [[[[1], [3]]], [[[9], [11]]], [[[2], [4]]], [[[10], [12]]],
     [[[5], [7]]], [[[13], [15]]], [[[6], [8]]], [[[14], [16]]]]
```

The output tensor has shape `[2, 2, 4, 1]` and value:

```
x = [[[[1], [3]], [[5], [7]]],
     [[[2], [4]], [[10], [12]]],
     [[[5], [7]], [[13], [15]]],
     [[[6], [8]], [[14], [16]]]]
```

`tf.space_to_depth(input, block_size, name=None)`

SpaceToDepth for tensors of type T.

Rearranges blocks of spatial data, into depth. More specifically, this op outputs a copy of the input tensor where values from the `height` and `width` dimensions are moved to the `depth` dimension. The attr `block_size` indicates the input block size and how the data is moved.

- Non-overlapping blocks of size `block_size x block size` are rearranged into depth at each location.
- The depth of the output tensor is `input_depth * block_size * block_size`.
- The input tensor's height and width must be divisible by block_size.

That is, assuming the input is in the shape: `[batch, height, width, depth]`, the shape of the output will be: `[batch, height/block_size, width/block_size, depth*block_size*block_size]`

This operation requires that the input tensor be of rank 4, and that `block_size` be >=1 and a divisor of both the input `height` and `width`.

This operation is useful for resizing the activations between convolutions (but keeping all data), e.g. instead of pooling. It is also useful for training purely convolutional models.

For example, given this input of shape `[1, 2, 2, 1]`, and block_size of 2:

```
x = [[[[1], [2]],
      [[3], [4]]]]
```

This operation will output a tensor of shape `[1, 1, 1, 4]`:

```
[[[[1, 2, 3, 4]]]]
```

Here, the input has a batch of 1 and each batch element has shape `[2, 2, 1]`, the corresponding output will have a single element (i.e. width and height are both 1) and will have a depth of 4 channels (1 * block_size * block_size). The output element shape is `[1, 1, 4]`.

For an input tensor with larger depth, here of shape `[1, 2, 2, 3]`, e.g.

```
x = [[[[1, 2, 3], [4, 5, 6]],
      [[7, 8, 9], [10, 11, 12]]]]
```

This operation, for block_size of 2, will return the following tensor of shape `[1, 1, 1, 12]`

```
[[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

Similarly, for the following input of shape `[1 4 4 1]`, and a block size of 2:

```
x = [[[[1],   [2],  [5],  [6]],
      [[3],   [4],  [7],  [8]],
      [[9],  [10], [13],  [14]],
      [[11], [12], [15],  [16]]]]
```

the operator will return the following tensor of shape `[1 2 2 4]`:

```
x = [[[[1, 2, 3, 4],
       [5, 6, 7, 8]],
      [[9, 10, 11, 12],
       [13, 14, 15, 16]]]]
```

Args:

- `input`: A `Tensor`.
- `block_size`: An `int` that is `>= 2`. The size of the spatial block.
- `name`: A name for the operation (optional).

Returns:

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

`tf.depth_to_space(input, block_size, name=None)`

DepthToSpace for tensors of type T.

Rearranges data from depth into blocks of spatial data. This is the reverse transformation of SpaceToDepth. More specifically, this op outputs a copy of the input tensor where values from the `depth` dimension are moved in spatial blocks to the `height` and `width` dimensions. The attr `block_size` indicates the input block size and how the data is moved.

- Chunks of data of size `block_size * block_size` from depth are rearranged into non-overlapping blocks of size `block_size x block_size`
- The width the output tensor is `input_depth * block_size`, whereas the height is `input_height * block_size`.
- The depth of the input tensor must be divisible by `block_size * block_size`.

That is, assuming the input is in the shape: `[batch, height, width, depth]`, the shape of the output will be: `[batch, height*block_size, width*block_size, depth/(block_size*block_size)]`

This operation requires that the input tensor be of rank 4, and that `block_size` be >=1 and that `block_size * block_size` be a divisor of the input depth.

This operation is useful for resizing the activations between convolutions (but keeping all data), e.g. instead of pooling. It is also useful for training purely convolutional models.

For example, given this input of shape `[1, 1, 1, 4]`, and a block size of 2:

```
x = [[[[1, 2, 3, 4]]]]
```

This operation will output a tensor of shape `[1, 2, 2, 1]`:

```
   [[[[1], [2]],
     [[3], [4]]]]
```

Here, the input has a batch of 1 and each batch element has shape `[1, 1, 4]`, the corresponding output will have 2x2 elements and will have a depth of 1 channel (1 = `4 / (block_size * block_size)`). The output element shape is `[2, 2, 1]`.

For an input tensor with larger depth, here of shape `[1, 1, 1, 12]`, e.g.

```
x = [[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

This operation, for block size of 2, will return the following tensor of shape `[1, 2, 2, 3]`

```
[[[[1, 2, 3], [4, 5, 6]],
  [[7, 8, 9], [10, 11, 12]]]]
```

Similarly, for the following input of shape `[1 2 2 4]`, and a block size of 2:

```
x =  [[[[1, 2, 3, 4],
       [5, 6, 7, 8]],
      [[9, 10, 11, 12],
       [13, 14, 15, 16]]]]
```

the operator will return the following tensor of shape `[1 4 4 1]`:

```
x = [[ [1],   [2],  [5],  [6]],
     [ [3],   [4],  [7],  [8]],
     [ [9],  [10], [13],  [14]],
     [ [11], [12], [15],  [16]]]
```

Args:

- `input`: A `Tensor`.
- `block_size`: An `int` that is `>= 2`. The size of the spatial block, same as in Space2Depth.
- `name`: A name for the operation (optional).

Returns:

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

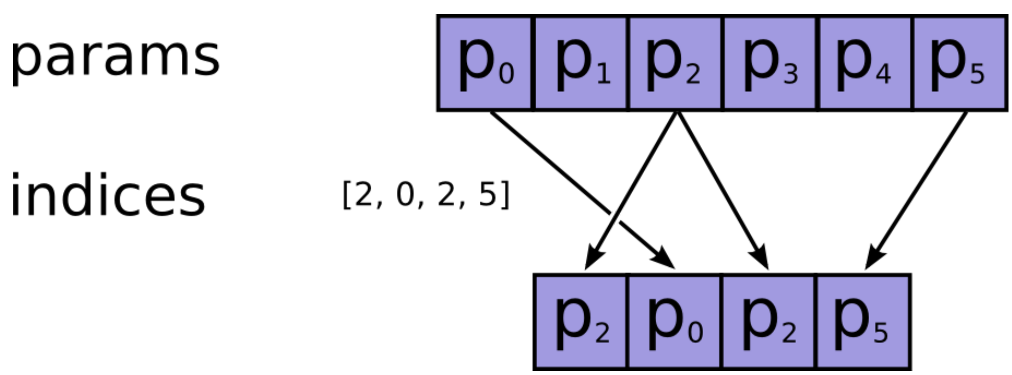
`tf.gather(params, indices, validate_indices=None, name=None)`

Gather slices from `params` according to `indices`.

`indices` must be an integer tensor of any dimension (usually 0-D or 1-D). Produces an output tensor with shape `indices.shape + params.shape[1:]` where:

```
# Scalar indices
output[:, ..., :] = params[indices, :, ... :]

# Vector indices
output[i, :, ..., :] = params[indices[i], :, ... :]

# Higher rank indices
output[i, ..., j, :, ... :] = params[indices[i, ..., j], :, ..., :]
```

If `indices` is a permutation and `len(indices) == params.shape[0]` then this operation will permute `params` accordingly.



Args:

- `params`: A `Tensor`.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- `validate_indices`: An optional `bool`. Defaults to `True`.
- `name`: A name for the operation (optional).

Returns:

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

`tf.gather_nd(params, indices, name=None)`

Gather values or slices from `params` according to `indices`.

`params` is a Tensor of rank `R` and `indices` is a Tensor of rank `M`.

`indices` must be integer tensor, containing indices into `params`. It must be shape `[d_0, ..., d_N, R]` where `0 < R <= M`.

The innermost dimension of `indices` (with length `R`) corresponds to indices into elements (if `R = M`) or slices (if `R < M`) along the `N`th dimension of `params`.

Produces an output tensor with shape

```
[d_0, ..., d_{n-1}, params.shape[R], ..., params.shape[M-1]].
```

Some examples below.

Simple indexing into a matrix:

```
indices = [[0, 0], [1, 1]]
params = [['a', 'b'], ['c', 'd']]
output = ['a', 'd']
```

Slice indexing into a matrix:

```
indices = [[1], [0]]
params = [['a', 'b'], ['c', 'd']]
output = [['c', 'd'], ['a', 'b']]
```

Indexing into a 3-tensor:

```
indices = [[1]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = [[['a1', 'b1'], ['c1', 'd1']]]


indices = [[0, 1], [1, 0]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = [['c0', 'd0'], ['a1', 'b1']]


indices = [[0, 0, 1], [1, 0, 1]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = ['b0', 'b1']
```

Batched indexing into a matrix:

```
indices = [[[0, 0]], [[0, 1]]]
params = [['a', 'b'], ['c', 'd']]
output = [['a'], ['b']]
```

Batched slice indexing into a matrix:

```
indices = [[[1]], [[0]]]
params = [['a', 'b'], ['c', 'd']]
output = [[['c', 'd']], [['a', 'b']]]
```

Batched indexing into a 3-tensor:

```
indices = [[[1]], [[0]]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = [[[['a1', 'b1'], ['c1', 'd1']]],
          [[['a0', 'b0'], ['c0', 'd0']]]]
```

```
indices = [[[0, 1], [1, 0]], [[0, 0], [1, 1]]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = [[['c0', 'd0'], ['a1', 'b1']],
          [['a0', 'b0'], ['c1', 'd1']]]


indices = [[[0, 0, 1], [1, 0, 1]], [[0, 1, 1], [1, 1, 0]]]
params = [[['a0', 'b0'], ['c0', 'd0']],
          [['a1', 'b1'], ['c1', 'd1']]]
output = [['b0', 'b1'], ['d0', 'c1']]
```

Args:

- `params`: A `Tensor`. `M-D`. The tensor from which to gather values.
- `indices`: A `Tensor`. Must be one of the following types: `int32`, `int64`. `(N+1)-D`. Index tensor having shape `[d_0, ..., d_N, R]`.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `params`. `(N+M-R)-D`. Values from `params` gathered from indices given by `indices`.

`tf.unique_with_counts(x, out_idx=None, name=None)`

Finds unique elements in a 1-D tensor.

This operation returns a tensor `y` containing all of the unique elements of `x` sorted in the same order that they occur in `x`. This operation also returns a tensor `idx` the same size as `x` that contains the index of each value of `x` in the unique output `y`. Finally, it returns a third tensor `count` that contains the count of each element of `y` in `x`. In other words:

```
y[idx[i]] = x[i] for i in [0, 1,...,rank(x) - 1]
```

For example:

```
# tensor 'x' is [1, 1, 2, 4, 4, 4, 7, 8, 8]
y, idx, count = unique_with_counts(x)
y ==> [1, 2, 4, 7, 8]
idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
count ==> [2, 1, 3, 1, 2]
```

Args:

- `x`: A `Tensor`. 1-D.
- `out_idx`: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int32`.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (y, idx, count).

- `y`: A `Tensor`. Has the same type as `x`. 1-D.
- `idx`: A `Tensor` of type `out_idx`. 1-D.
- `count`: A `Tensor` of type `out_idx`. 1-D.

`tf.dynamic_partition(data, partitions, num_partitions, name=None)`

Partitions `data` into `num_partitions` tensors using indices from `partitions`.

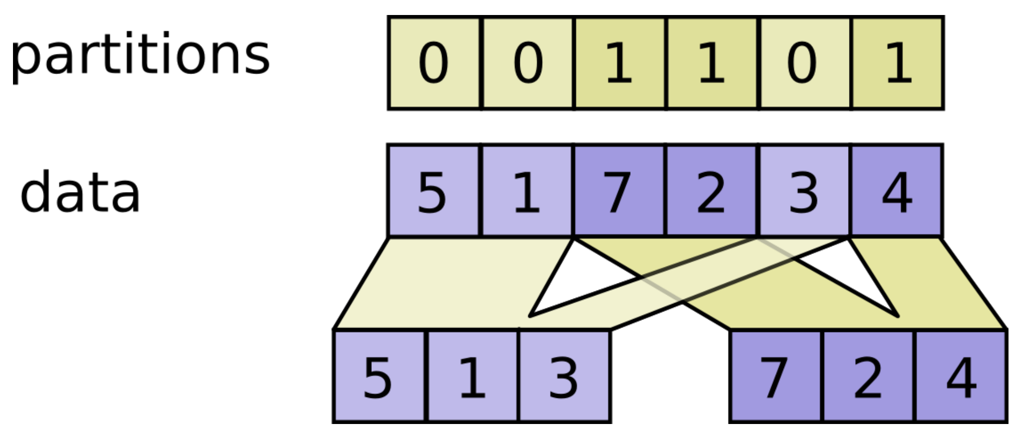
For each index tuple `js` of size `partitions.ndim`, the slice `data[js, ...]` becomes part of `outputs[partitions[js]]`. The slices with `partitions[js] = i` are placed in `outputs[i]` in lexicographic order of `js`, and the first dimension of `outputs[i]` is the number of entries in `partitions` equal to `i`. In detail,

```
outputs[i].shape = [sum(partitions == i)] + data.shape[partitions.ndim:]

outputs[i] = pack([data[js, ...] for js if partitions[js] == i])
```

`data.shape` must start with `partitions.shape`.

For example:

```
# Scalar partitions
partitions = 1
num_partitions = 2
data = [10, 20]
outputs[0] = []  # Empty with shape [0, 2]
outputs[1] = [[10, 20]]

# Vector partitions
partitions = [0, 0, 1, 1, 0]
num_partitions = 2
data = [10, 20, 30, 40, 50]
outputs[0] = [10, 20, 50]
outputs[1] = [30, 40]
```



Args:

- `data`: A `Tensor`.
- `partitions`: A `Tensor` of type `int32`. Any shape. Indices in the range `[0, num_partitions)`.
- `num_partitions`: An `int` that is `>= 1`. The number of partitions to output.

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

Returns:

A list of `num_partitions` `Tensor` objects of the same type as data.

`tf.dynamic_stitch(indices, data, name=None)`

Interleave the values from the `data` tensors into a single tensor.

Builds a merged tensor such that

```
merged[indices[m][i, ..., j], ...] = data[m][i, ..., j, ...]
```

For example, if each `indices[m]` is scalar or vector, we have

```
# Scalar indices
merged[indices[m], ...] = data[m][...]

# Vector indices
merged[indices[m][i], ...] = data[m][i, ...]
```

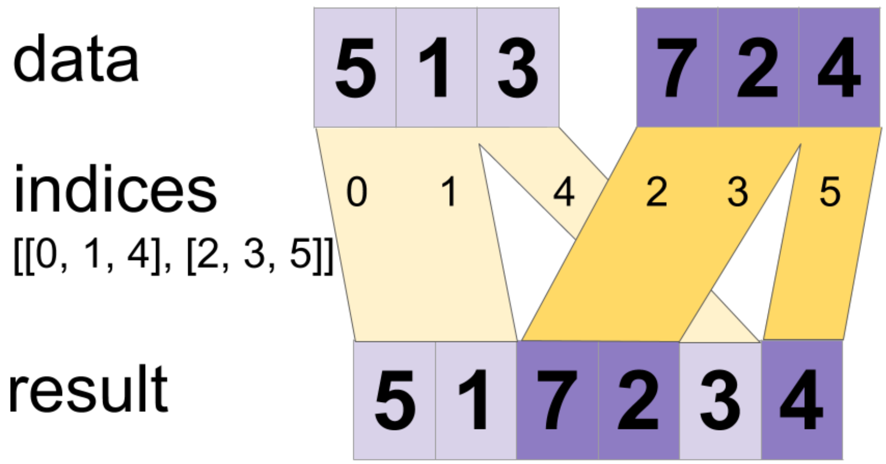
Each `data[i].shape` must start with the corresponding `indices[i].shape`, and the rest of `data[i].shape` must be constant w.r.t. `i`. That is, we must have `data[i].shape = indices[i].shape + constant`. In terms of this `constant`, the output shape is

```
merged.shape = [max(indices)] + constant
```

Values are merged in order, so if an index appears in both `indices[m][i]` and `indices[n][j]` for `(m,i) < (n,j)` the slice `data[n][j]` will appear in the merged result.

For example:

```
indices[0] = 6
indices[1] = [4, 1]
indices[2] = [[5, 2], [0, 3]]
data[0] = [61, 62]
data[1] = [[41, 42], [11, 12]]
data[2] = [[[51, 52], [21, 22]], [[1, 2], [31, 32]]]
merged = [[1, 2], [11, 12], [21, 22], [31, 32], [41, 42],
          [51, 52], [61, 62]]
```



Args:

- `indices`: A list of at least 1 `Tensor` objects of type `int32`.
- `data`: A list with the same number of `Tensor` objects as `indices` of `Tensor` objects of the same type.
- `name`: A name for the operation (optional).

Returns:

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

`tf.boolean_mask(tensor, mask, name='boolean_mask')`

Apply boolean mask to tensor. Numpy equivalent is `tensor[mask]`.

```
# 1-D example
tensor = [0, 1, 2, 3]
mask = [True, False, True, False]
boolean_mask(tensor, mask) ==> [0, 2]
```

In general, `0 < dim(mask) = K <= dim(tensor)`, and `mask`'s shape must match the first K dimensions of `tensor`'s shape. We then have: `boolean_mask(tensor, mask)[i, j1,...,jd] = tensor[i1,...,iK,j1,...,jd]` where `(i1,...,iK)` is the ith `True` entry of `mask` (row-major order).

Args:

- `tensor`: N-D tensor.
- `mask`: K-D boolean tensor, K <= N and K must be known statically.
- `name`: A name for this operation (optional).

Returns:

Tensor populated by entries in `tensor` corresponding to `True` values in `mask`.

Raises:

`ValueError`: If shapes do not conform.

`Examples`:

```
# 2-D example
tensor = [[1, 2], [3, 4], [5, 6]]
```

```
mask = [True, False, True]
boolean_mask(tensor, mask) ==> [[1, 2], [5, 6]]
```

---

tf.one_hot(indices, depth, on_value=None, off_value=None, axis=None, dtype=None, name=None)

Returns a one-hot tensor.

The locations represented by indices in `indices` take value `on_value`, while all other locations take value `off_value`.

`on_value` and `off_value` must have matching data types. If `dtype` is also provided, they must be the same data type as specified by `dtype`.

If `on_value` is not provided, it will default to the value `1` with type `dtype`

If `off_value` is not provided, it will default to the value `0` with type `dtype`

If the input `indices` is rank `N`, the output will have rank `N+1`. The new axis is created at dimension `axis` (default: the new axis is appended at the end).

If `indices` is a scalar the output shape will be a vector of length `depth`

If `indices` is a vector of length `features`, the output shape will be: `features x depth if axis == -1 depth x features if axis == 0`

If `indices` is a matrix (batch) with shape `[batch, features]`, the output shape will be: `batch x features x depth if axis == -1 batch x depth x features if axis == 1 depth x batch x features if axis == 0`

If `dtype` is not provided, it will attempt to assume the data type of `on_value` or `off_value`, if one or both are passed in. If none of `on_value`, `off_value`, or `dtype` are provided, `dtype` will default to the value `tf.float32`

Note: If a non-numeric data type output is desired (tf.string, tf.bool, etc.), both `on_value` and `off_value` must be provided to `one_hot`

Examples

Suppose that

```
indices = [0, 2, -1, 1]
depth = 3
on_value = 5.0
off_value = 0.0
axis = -1
```

Then output is `[4 x 3]`:

```
output =
[5.0 0.0 0.0]  // one_hot(0)
[0.0 0.0 5.0]  // one_hot(2)
[0.0 0.0 0.0]  // one_hot(-1)
[0.0 5.0 0.0]  // one_hot(1)
```

Suppose that

```
indices = [[0, 2], [1, -1]]
depth = 3
on_value = 1.0
off_value = 0.0
axis = -1
```

Then output is `[2 x 2 x 3]`:

```
output =
[
  [1.0, 0.0, 0.0]  // one_hot(0)
  [0.0, 0.0, 1.0]  // one_hot(2)
][
  [0.0, 1.0, 0.0]  // one_hot(1)
  [0.0, 0.0, 0.0]  // one_hot(-1)
]
```

Using default values for `on_value` and `off_value`:

```
indices = [0, 1, 2]
depth = 3
```

The output will be

```
output =
[[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]]
```

Args:

- `indices`: A `Tensor` of indices.
- `depth`: A scalar defining the depth of the one hot dimension.
- `on_value`: A scalar defining the value to fill in output when `indices[j] = i`. (default: 1)
- `off_value`: A scalar defining the value to fill in output when `indices[j] != i`. (default: 0)
- `axis`: The axis to fill (default: -1, a new inner-most axis).
- `dtype`: The data type of the output tensor.

Returns:

- `output`: The one-hot tensor.

Raises:

- `TypeError`: If dtype of either `on_value` or `off_value` don't match `dtype`
- `TypeError`: If dtype of `on_value` and `off_value` don't match one another

`tf.sequence_mask(lengths, maxlen=None, dtype=tf.bool, name=None)`

Return a mask tensor representing the first N positions of each row.

Example: `python tf.sequence_mask([1, 3, 2], 5) = [[True, False, False, False, False], [True, True, True, False, False], [True, True, False, False, False]]`

Args:

- `lengths`: 1D integer tensor, all its values < maxlen.
- `maxlen`: scalar integer tensor, maximum length of each row. Default: use maximum over lengths.
- `dtype`: output type of the resulting tensor.
- `name`: name of the op.

Returns:

A 2D mask tensor, as shown in the example above, cast to specified dtype.

Raises:

- `ValueError`: if the arguments have invalid rank.