Module: tf.nn / tf.compat.v1.nn

- **Contents**
- Modules
- Functions
- Other Members

Wrappers for primitive Neural Net (NN) Operations.

Modules

`rnn_cell` module: Module for constructing RNN Cells.

Functions

`all_candidate_sampler(...)`: Generate the set of all classes.
`atrous_conv2d(...)`: Atrous convolution (a.k.a. convolution with holes or dilated convolution).
`atrous_conv2d_transpose(...)`: The transpose of `atrous_conv2d`.
`avg_pool(...)`: Performs the average pooling on the input.
`avg_pool1d(...)`: Performs the average pooling on the input.
`avg_pool2d(...)`: Performs the average pooling on the input.
`avg_pool3d(...)`: Performs the average pooling on the input.
`avg_pool_v2(...)`: Performs the avg pooling on the input.
`batch_norm_with_global_normalization(...)`: Batch normalization.
`batch_normalization(...)`: Batch normalization.
`bias_add(...)`: Adds `bias` to `value`.
`bidirectional_dynamic_rnn(...)`: Creates a dynamic version of bidirectional recurrent neural network. (deprecated)
`collapse_repeated(...)`: Merge repeated labels into single labels.
`compute_accidental_hits(...)`: Compute the position ids in `sampled_candidates` matching `true_classes`.
`compute_average_loss(...)`: Scales per-example losses with sample_weights and computes their average.
`conv1d(...)`: Computes a 1-D convolution given 3-D input and filter tensors. (deprecated argument values) (deprecated argument values)
`conv1d_transpose(...)`: The transpose of `conv1d`.
`conv2d(...)`: Computes a 2-D convolution given 4-D `input` and `filter` tensors.
`conv2d_backprop_filter(...)`: Computes the gradients of convolution with respect to the filter.
`conv2d_backprop_input(...)`: Computes the gradients of convolution with respect to the input.
`conv2d_transpose(...)`: The transpose of `conv2d`.
`conv3d(...)`: Computes a 3-D convolution given 5-D `input` and `filter` tensors.
`conv3d_backprop_filter(...)`: Computes the gradients of 3-D convolution with respect to the filter.
`conv3d_backprop_filter_v2(...)`: Computes the gradients of 3-D convolution with respect to the filter.
`conv3d_transpose(...)`: The transpose of `conv3d`.
`conv_transpose(...)`: The transpose of `convolution`.
`convolution(...)`: Computes sums of N-D convolutions (actually cross-correlation).
`crelu(...)`: Computes Concatenated ReLU.
`ctc_beam_search_decoder(...)`: Performs beam search decoding on the logits given in input.
`ctc_beam_search_decoder_v2(...)`: Performs beam search decoding on the logits given in input.
`ctc_greedy_decoder(...)`: Performs greedy decoding on the logits given in input (best path).
`ctc_loss(...)`: Computes the CTC (Connectionist Temporal Classification) Loss.
`ctc_loss_v2(...)`: Computes CTC (Connectionist Temporal Classification) loss.

`ctc_unique_labels(...)`: Get unique labels and indices for batched labels for `tf.nn.ctc_loss`.
`depth_to_space(...)`: DepthToSpace for tensors of type T.
`depthwise_conv2d(...)`: Depthwise 2-D convolution.
`depthwise_conv2d_backprop_filter(...)`: Computes the gradients of depthwise convolution with respect to the filter.
`depthwise_conv2d_backprop_input(...)`: Computes the gradients of depthwise convolution with respect to the input.
`depthwise_conv2d_native(...)`: Computes a 2-D depthwise convolution given 4-D `input` and `filter` tensors.
`depthwise_conv2d_native_backprop_filter(...)`: Computes the gradients of depthwise convolution with respect to the filter.
`depthwise_conv2d_native_backprop_input(...)`: Computes the gradients of depthwise convolution with respect to the input.
`dilation2d(...)`: Computes the grayscale dilation of 4-D `input` and 3-D `filter` tensors.
`dropout(...)`: Computes dropout. (deprecated arguments)
`dynamic_rnn(...)`: Creates a recurrent neural network specified by RNNCell `cell`. (deprecated)
`elu(...)`: Computes exponential linear: `exp(features) - 1` if < 0, `features` otherwise.
`embedding_lookup(...)`: Looks up `ids` in a list of embedding tensors.
`embedding_lookup_sparse(...)`: Computes embeddings for the given ids and weights.
`erosion2d(...)`: Computes the grayscale erosion of 4-D `value` and 3-D `kernel` tensors.
`fixed_unigram_candidate_sampler(...)`: Samples a set of classes using the provided (fixed) base distribution.
`fractional_avg_pool(...)`: Performs fractional average pooling on the input. (deprecated)
`fractional_max_pool(...)`: Performs fractional max pooling on the input. (deprecated)
`fused_batch_norm(...)`: Batch normalization.
`in_top_k(...)`: Says whether the targets are in the top `K` predictions.
`l2_loss(...)`: L2 Loss.
`l2_normalize(...)`: Normalizes along dimension `axis` using an L2 norm. (deprecated arguments)
`leaky_relu(...)`: Compute the Leaky ReLU activation function.
`learned_unigram_candidate_sampler(...)`: Samples a set of classes from a distribution learned during training.
`local_response_normalization(...)`: Local Response Normalization.
`log_poisson_loss(...)`: Computes log Poisson loss given `log_input`.
`log_softmax(...)`: Computes log softmax activations. (deprecated arguments)
`log_uniform_candidate_sampler(...)`: Samples a set of classes using a log-uniform (Zipfian) base distribution.
`lrn(...)`: Local Response Normalization.
`max_pool(...)`: Performs the max pooling on the input.
`max_pool1d(...)`: Performs the max pooling on the input.
`max_pool2d(...)`: Performs the max pooling on the input.
`max_pool3d(...)`: Performs the max pooling on the input.
`max_pool_v2(...)`: Performs the max pooling on the input.
`max_pool_with_argmax(...)`: Performs max pooling on the input and outputs both max values and indices.
`moments(...)`: Calculate the mean and variance of `x`.
`nce_loss(...)`: Computes and returns the noise-contrastive estimation training loss.
`normalize_moments(...)`: Calculate the mean and variance of based on the sufficient statistics.
`pool(...)`: Performs an N-D pooling operation.
`quantized_avg_pool(...)`: Produces the average pool of the input tensor for quantized types.
`quantized_conv2d(...)`: Computes a 2D convolution given quantized 4D input and filter tensors.
`quantized_max_pool(...)`: Produces the max pool of the input tensor for quantized types.

`quantized_relu_x(...)`: Computes Quantized Rectified Linear X: `min(max(features, 0), max_value)`
`raw_rnn(...)`: Creates an `RNN` specified by RNNCell `cell` and loop function `loop_fn`.
`relu(...)`: Computes rectified linear: `max(features, 0)`.
`relu6(...)`: Computes Rectified Linear 6: `min(max(features, 0), 6)`.
`relu_layer(...)`: Computes Relu(x * weight + biases).
`safe_embedding_lookup_sparse(...)`: Lookup embedding results, accounting for invalid IDs and empty features.
`sampled_softmax_loss(...)`: Computes and returns the sampled softmax training loss.
`scale_regularization_loss(...)`: Scales the sum of the given regularization losses by number of replicas.
`selu(...)`: Computes scaled exponential linear: `scale * alpha * (exp(features) - 1)`
`separable_conv2d(...)`: 2-D convolution with separable filters.
`sigmoid(...)`: Computes sigmoid of `x` element-wise.
`sigmoid_cross_entropy_with_logits(...)`: Computes sigmoid cross entropy given `logits`.
`softmax(...)`: Computes softmax activations. (deprecated arguments)
`softmax_cross_entropy_with_logits(...)`: Computes softmax cross entropy between `logits`and `labels`. (deprecated)
`softmax_cross_entropy_with_logits_v2(...)`: Computes softmax cross entropy between `logits` and `labels`. (deprecated arguments)
`softplus(...)`: Computes softplus: `log(exp(features) + 1)`.
`softsign(...)`: Computes softsign: `features / (abs(features) + 1)`.
`space_to_batch(...)`: SpaceToBatch for 4-D tensors of type T.
`space_to_depth(...)`: SpaceToDepth for tensors of type T.
`sparse_softmax_cross_entropy_with_logits(...)`: Computes sparse softmax cross entropy between `logits` and `labels`.
`static_bidirectional_rnn(...)`: Creates a bidirectional recurrent neural network. (deprecated)
`static_rnn(...)`: Creates a recurrent neural network specified by RNNCell `cell`. (deprecated)
`static_state_saving_rnn(...)`: RNN that accepts a state saver for time-truncated RNN calculation. (deprecated)
`sufficient_statistics(...)`: Calculate the sufficient statistics for the mean and variance of `x`.
`tanh(...)`: Computes hyperbolic tangent of `x` element-wise.
`top_k(...)`: Finds values and indices of the `k` largest entries for the last dimension.
`uniform_candidate_sampler(...)`: Samples a set of classes using a uniform base distribution.
`weighted_cross_entropy_with_logits(...)`: Computes a weighted cross entropy. (deprecated arguments)
`weighted_moments(...)`: Returns the frequency-weighted mean and variance of `x`.
`with_space_to_batch(...)`: Performs `op` on the space-to-batch representation of `input`.
`xw_plus_b(...)`: Computes matmul(x, weights) + biases.
`zero_fraction(...)`: Returns the fraction of zeros in `value`.

Other Members

- `swish`

tf.nn.atrous_conv2d

- Contents
- Aliases:

Atrous convolution (a.k.a. convolution with holes or dilated convolution).

Aliases:

- `tf.compat.v1.nn.atrous_conv2d`

- `tf.compat.v2.nn.atrous_conv2d`
- `tf.nn.atrous_conv2d`

```
tf.nn.atrous_conv2d(
    value,
    filters,
    rate,
    padding,
    name=None
)

```

Defined in `python/ops/nn_ops.py`.

This function is a simpler wrapper around the more general `tf.nn.convolution`, and exists only for backwards compatibility. You can use `tf.nn.convolution` to perform 1-D, 2-D, or 3-D atrous convolution.

Computes a 2-D atrous convolution, also known as convolution with holes or dilated convolution, given 4-D `value` and `filters` tensors. If the `rate` parameter is equal to one, it performs regular 2-D convolution. If the `rate` parameter is greater than one, it performs convolution with holes, sampling the input values every `rate` pixels in the `height` and `width` dimensions. This is equivalent to convolving the input with a set of upsampled filters, produced by inserting `rate - 1` zeros between two consecutive values of the filters along the `height` and `width` dimensions, hence the name atrous convolution or convolution with holes (the French word trous means holes in English).

More specifically:

```
output[batch, height, width, out_channel] =
    sum_{dheight, dwidth, in_channel} (
        filters[dheight, dwidth, in_channel, out_channel] *
        value[batch, height + rate*dheight, width + rate*dwidth, in_channel]
    )
```

Atrous convolution allows us to explicitly control how densely to compute feature responses in fully convolutional networks. Used in conjunction with bilinear interpolation, it offers an alternative to `conv2d_transpose` in dense prediction tasks such as semantic image segmentation, optical flow computation, or depth estimation. It also allows us to effectively enlarge the field of view of filters without increasing the number of parameters or the amount of computation.

For a description of atrous convolution and how it can be used for dense feature extraction, please see: [Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs](). The same operation is investigated further in [Multi-Scale Context Aggregation by Dilated Convolutions](). Previous works that effectively use atrous convolution in different ways are, among others, [OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks]() and [Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks](). Atrous convolution is also closely related to the so-called noble identities in multi-rate signal processing.

There are many different ways to implement atrous convolution (see the refs above). The implementation here reduces

```
    atrous_conv2d(value, filters, rate, padding=padding)
```

to the following three operations:

```
    paddings = ...
    net = space_to_batch(value, paddings, block_size=rate)
    net = conv2d(net, filters, strides=[1, 1, 1, 1], padding="VALID")
    crops = ...
```

```
    net = batch_to_space(net, crops, block_size=rate)
```

Advanced usage. Note the following optimization: A sequence of `atrous_conv2d` operations with identical `rate` parameters, 'SAME' `padding`, and filters with odd heights/ widths:

```
    net = atrous_conv2d(net, filters1, rate, padding="SAME")
    net = atrous_conv2d(net, filters2, rate, padding="SAME")
    ...
    net = atrous_conv2d(net, filtersK, rate, padding="SAME")
```

can be equivalently performed cheaper in terms of computation and memory as:

```
    pad = ...  # padding so that the input dims are multiples of rate
    net = space_to_batch(net, paddings=pad, block_size=rate)
    net = conv2d(net, filters1, strides=[1, 1, 1, 1], padding="SAME")
    net = conv2d(net, filters2, strides=[1, 1, 1, 1], padding="SAME")
    ...
    net = conv2d(net, filtersK, strides=[1, 1, 1, 1], padding="SAME")
    net = batch_to_space(net, crops=pad, block_size=rate)
```

because a pair of consecutive `space_to_batch` and `batch_to_space` ops with the same `block_size` cancel out when their respective `paddings` and `crops` inputs are identical.

Args:

- **`value`**: A 4-D `Tensor` of type `float`. It needs to be in the default "NHWC" format. Its shape is `[batch, in_height, in_width, in_channels]`.
- **`filters`**: A 4-D `Tensor` with the same type as `value` and shape `[filter_height, filter_width, in_channels, out_channels]`. `filters`' `in_channels` dimension must match that of `value`. Atrous convolution is equivalent to standard convolution with upsampled filters with effective height `filter_height + (filter_height - 1) * (rate - 1)` and effective width `filter_width + (filter_width - 1) * (rate - 1)`, produced by inserting `rate - 1` zeros along consecutive elements across the `filters`' spatial dimensions.
- **`rate`**: A positive int32. The stride with which we sample input values across the `height` and `width` dimensions. Equivalently, the rate by which we upsample the filter values by inserting zeros across the `height` and `width` dimensions. In the literature, the same parameter is sometimes called `input stride` or `dilation`.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- **`name`**: Optional name for the returned tensor.

Returns:

A `Tensor` with the same type as `value`. Output shape with `'VALID'` padding is:

```
[batch, height - 2 * (filter_width - 1),
 width - 2 * (filter_height - 1), out_channels].
```

Output shape with `'SAME'` padding is:

```
[batch, height, width, out_channels].
```

Raises:

`ValueError`: If input/output depth does not match `filters`' shape, or if padding is other than `'VALID'` or `'SAME'`.

tf.nn.atrous_conv2d_transpose

- Contents
- Aliases:

The transpose of `atrous_conv2d`.

Aliases:

- `tf.compat.v1.nn.atrous_conv2d_transpose`
- `tf.compat.v2.nn.atrous_conv2d_transpose`
- `tf.nn.atrous_conv2d_transpose`

```
tf.nn.atrous_conv2d_transpose(
    value,
    filters,
    output_shape,
    rate,
    padding,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is really the transpose (gradient) of `atrous_conv2d` rather than an actual deconvolution.

Args:

- **`value`**: A 4-D `Tensor` of type `float`. It needs to be in the default `NHWC` format. Its shape is `[batch, in_height, in_width, in_channels]`.
- **`filters`**: A 4-D `Tensor` with the same type as `value` and shape `[filter_height, filter_width, out_channels, in_channels]`. `filters`' `in_channels` dimension must match that of `value`. Atrous convolution is equivalent to standard convolution with upsampled filters with effective height `filter_height + (filter_height - 1) * (rate - 1)` and effective width `filter_width + (filter_width - 1) * (rate - 1)`, produced by inserting `rate - 1` zeros along consecutive elements across the `filters`' spatial dimensions.
- **`output_shape`**: A 1-D `Tensor` of shape representing the output shape of the deconvolution op.
- **`rate`**: A positive int32. The stride with which we sample input values across the `height` and `width` dimensions. Equivalently, the rate by which we upsample the filter values by inserting zeros across the `height` and `width` dimensions. In the literature, the same parameter is sometimes called `input stride` or `dilation`.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm.
- **`name`**: Optional name for the returned tensor.

Returns:

A `Tensor` with the same type as `value`.

Raises:

- **`ValueError`**: If input/output depth does not match `filters`' shape, or if padding is other than `'VALID'` or `'SAME'`, or if the `rate` is less than one, or if the output_shape is not a tensor with 4 elements.

tf.nn.avg_pool

- Contents
- Aliases:

Performs the avg pooling on the input.

Aliases:

- `tf.compat.v1.nn.avg_pool_v2`

- `tf.compat.v2.nn.avg_pool`
- `tf.nn.avg_pool`

```
tf.nn.avg_pool(
    input,
    ksize,
    strides,
    padding,
    data_format=None,
    name=None
)

```

Defined in `python/ops/nn_ops.py`.

Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.

Args:

- **`input`**: Tensor of rank N+2, of shape `[batch_size] + input_spatial_shape + [num_channels]` if `data_format` does not start with "NC" (default), or `[batch_size, num_channels] + input_spatial_shape` if data_format starts with "NC". Pooling happens over the spatial dimensions only.
- **`ksize`**: An int or list of `ints` that has length `1`, `N` or `N+2`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `N` or `N+2`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. Specifies the channel dimension. For N=1 it can be either "NWC" (default) or "NCW", for N=2 it can be either "NHWC" (default) or "NCHW" and for N=3 either "NDHWC" (default) or "NCDHW".
- **`name`**: Optional name for the operation.

Returns:

A `Tensor` of format specified by `data_format`. The average pooled output tensor.

tf.nn.avg_pool1d

- Contents
- Aliases:

Performs the average pooling on the input.

Aliases:

- `tf.compat.v1.nn.avg_pool1d`
- `tf.compat.v2.nn.avg_pool1d`
- `tf.nn.avg_pool1d`

```
tf.nn.avg_pool1d(
    input,
    ksize,
    strides,
    padding,
    data_format='NWC',
    name=None
)

```

Defined in `python/ops/nn_ops.py`.
Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.
Note internally this op reshapes and uses the underlying 2d operation.

Args:

- **`input`**: A 3-D `Tensor` of the format specified by `data_format`.
- **`ksize`**: An int or list of `ints` that has length `1` or `3`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1` or `3`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: An optional string from: "NWC", "NCW". Defaults to "NWC".
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.avg_pool1d

- Contents
- Aliases:

Performs the average pooling on the input.

Aliases:

- `tf.compat.v1.nn.avg_pool1d`
- `tf.compat.v2.nn.avg_pool1d`
- `tf.nn.avg_pool1d`

```
tf.nn.avg_pool1d(
    input,
    ksize,
    strides,
    padding,
    data_format='NWC',
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.
Note internally this op reshapes and uses the underlying 2d operation.

Args:

- **`input`**: A 3-D `Tensor` of the format specified by `data_format`.
- **`ksize`**: An int or list of `ints` that has length `1` or `3`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1` or `3`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: An optional string from: "NWC", "NCW". Defaults to "NWC".
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.avg_pool3d

- Contents
- Aliases:

Performs the average pooling on the input.

Aliases:

- `tf.compat.v1.nn.avg_pool3d`
- `tf.compat.v2.nn.avg_pool3d`
- `tf.nn.avg_pool3d`

```
tf.nn.avg_pool3d(
    input,
    ksize,
    strides,
    padding,
    data_format='NDHWC',
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.

Args:

- **`input`**: A 5-D `Tensor` of shape `[batch, height, width, channels]` and type `float32`, `float64`, `qint8`, `quint8`, or `qint32`.
- **`ksize`**: An int or list of `ints` that has length `1`, `3` or `5`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `3` or `5`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. 'NDHWC' and 'NCDHW' are supported.
- **`name`**: Optional name for the operation.

Returns:

A `Tensor` with the same type as `value`. The average pooled output tensor.

tf.nn.batch_normalization

- Contents
- Aliases:

Batch normalization.

Aliases:

- `tf.compat.v1.nn.batch_normalization`
- `tf.compat.v2.nn.batch_normalization`
- `tf.nn.batch_normalization`

```
tf.nn.batch_normalization(
    x,
    mean,
    variance,
    offset,
```

```
    scale,
    variance_epsilon,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

Normalizes a tensor by `mean` and `variance`, and applies (optionally) a `scale` γ to it, as well as an `offset` β:

γ(x−μ)σ+β

`mean`, `variance`, `offset` and `scale` are all expected to be of one of two shapes:

- In all generality, they can have the same number of dimensions as the input `x`, with identical sizes as `x` for the dimensions that are not normalized over (the 'depth' dimension(s)), and dimension 1 for the others which are being normalized over. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=True)` during training, or running averages thereof during inference.
- In the common case where the 'depth' dimension is the last dimension in the input tensor `x`, they may be one dimensional tensors of the same size as the 'depth' dimension. This is the case for example for the common `[batch, depth]` layout of fully-connected layers, and `[batch, height, width, depth]` for convolutions. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=False)` during training, or running averages thereof during inference.

See Source: [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift; S. Ioffe, C. Szegedy]().

Args:

- **`x`**: Input `Tensor` of arbitrary dimensionality.
- **`mean`**: A mean `Tensor`.
- **`variance`**: A variance `Tensor`.
- **`offset`**: An offset `Tensor`, often denoted β in equations, or None. If present, will be added to the normalized tensor.
- **`scale`**: A scale `Tensor`, often denoted γ in equations, or `None`. If present, the scale is applied to the normalized tensor.
- **`variance_epsilon`**: A small float number to avoid dividing by 0.
- **`name`**: A name for this operation (optional).

Returns:

the normalized, scaled, offset tensor.

tf.nn.batch_norm_with_global_normalization

- Contents
- Aliases:

Batch normalization.

Aliases:

- `tf.compat.v2.nn.batch_norm_with_global_normalization`
- `tf.nn.batch_norm_with_global_normalization`

```
tf.nn.batch_norm_with_global_normalization(
    input,
    mean,
    variance,
    beta,
    gamma,
```

```
    variance_epsilon,
    scale_after_normalization,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.
This op is deprecated. See `tf.nn.batch_normalization`.

Args:

- **`input`**: A 4D input Tensor.
- **`mean`**: A 1D mean Tensor with size matching the last dimension of t. This is the first output from tf.nn.moments, or a saved moving average thereof.
- **`variance`**: A 1D variance Tensor with size matching the last dimension of t. This is the second output from tf.nn.moments, or a saved moving average thereof.
- **`beta`**: A 1D beta Tensor with size matching the last dimension of t. An offset to be added to the normalized tensor.
- **`gamma`**: A 1D gamma Tensor with size matching the last dimension of t. If "scale_after_normalization" is true, this tensor will be multiplied with the normalized tensor.
- **`variance_epsilon`**: A small float number to avoid dividing by 0.
- **`scale_after_normalization`**: A bool indicating whether the resulted tensor needs to be multiplied with gamma.
- **`name`**: A name for this operation (optional).

Returns:

A batch-normalized `t`.

tf.nn.bias_add

- Contents
- Aliases:

Adds `bias` to `value`.

Aliases:

- `tf.compat.v1.nn.bias_add`
- `tf.compat.v2.nn.bias_add`
- `tf.nn.bias_add`

```
tf.nn.bias_add(
    value,
    bias,
    data_format=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This is (mostly) a special case of `tf.add` where `bias` is restricted to 1-D. Broadcasting is supported, so `value` may have any number of dimensions. Unlike `tf.add`, the type of `bias` is allowed to differ from `value` in the case where both types are quantized.

Args:

- **`value`**: A `Tensor` with type `float`, `double`, `int64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, or `complex128`.
- **`bias`**: A 1-D `Tensor` with size matching the last dimension of `value`. Must be the same type as `value` unless `value` is a quantized type, in which case a different quantized type may be used.

- **`data_format`**: A string. 'N...C' and 'NC...' are supported.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` with the same type as `value`.

tf.nn.collapse_repeated

- Contents
- Aliases:

Merge repeated labels into single labels.

Aliases:

- `tf.compat.v1.nn.collapse_repeated`
- `tf.compat.v2.nn.collapse_repeated`
- `tf.nn.collapse_repeated`

```
tf.nn.collapse_repeated(
    labels,
    seq_length,
    name=None
)
```

Defined in `python/ops/ctc_ops.py`.

Args:

- **`labels`**: Tensor of shape [batch, max value in seq_length]
- **`seq_length`**: Tensor of shape [batch], sequence length of each batch element.
- **`name`**: A name for this `Op`. Defaults to "collapse_repeated_labels".

Returns:
A tuple `(collapsed_labels, new_seq_length)` where

- **`collapsed_labels`**: Tensor of shape [batch, max_seq_length] with repeated labels collapsed and padded to max_seq_length, eg: `[[A, A, B, B, A], [A, B, C, D, E]] => [[A, B, A, 0, 0], [A, B, C, D, E]]`
- **`new_seq_length`**: int tensor of shape [batch] with new sequence lengths.

tf.nn.compute_accidental_hits

- Contents
- Aliases:

Compute the position ids in `sampled_candidates` matching `true_classes`.

Aliases:

- `tf.compat.v1.nn.compute_accidental_hits`
- `tf.compat.v2.nn.compute_accidental_hits`
- `tf.nn.compute_accidental_hits`

```
tf.nn.compute_accidental_hits(
    true_classes,
    sampled_candidates,
    num_true,
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.
In Candidate Sampling, this operation facilitates virtually removing sampled classes which happen to match target classes. This is done in Sampled Softmax and Sampled Logistic.
See our [Candidate Sampling Algorithms Reference](#).
We presuppose that the `sampled_candidates` are unique.
We call it an 'accidental hit' when one of the target classes matches one of the sampled classes.
This operation reports accidental hits as triples `(index, id, weight)`, where `index` represents the row number in `true_classes`, `id` represents the position in `sampled_candidates`, and weight is `-FLOAT_MAX`.
The result of this op should be passed through a `sparse_to_dense` operation, then added to the logits of the sampled classes. This removes the contradictory effect of accidentally sampling the true target classes as noise classes for the same example.

Args:

- **`true_classes`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- **`sampled_candidates`**: A tensor of type `int64` and shape `[num_sampled]`. The sampled_candidates output of CandidateSampler.
- **`num_true`**: An `int`. The number of target classes per training example.
- **`seed`**: An `int`. An operation-specific seed. Default is 0.
- **`name`**: A name for the operation (optional).

Returns:

- **`indices`**: A `Tensor` of type `int32` and shape `[num_accidental_hits]`. Values indicate rows in `true_classes`.
- **`ids`**: A `Tensor` of type `int64` and shape `[num_accidental_hits]`. Values indicate positions in `sampled_candidates`.
- **`weights`**: A `Tensor` of type `float` and shape `[num_accidental_hits]`. Each value is `-FLOAT_MAX`.

tf.nn.compute_average_loss

- Contents
- Aliases:
- Used in the tutorials:

Scales per-example losses with sample_weights and computes their average.

Aliases:

- `tf.compat.v1.nn.compute_average_loss`
- `tf.compat.v2.nn.compute_average_loss`
- `tf.nn.compute_average_loss`

```
tf.nn.compute_average_loss(
    per_example_loss,
    sample_weight=None,
    global_batch_size=None
)
```

Defined in `python/ops/nn_impl.py`.

Used in the tutorials:

- [tf.distribute.Strategy with training loops](#)

Usage with distribution strategy and custom training loop:

```
with strategy.scope():
  def compute_loss(labels, predictions, sample_weight=None):
```

```
    # If you are using a `Loss` class instead, set reduction to `NONE` so that
    # we can do the reduction afterwards and divide by global batch size.
    per_example_loss = tf.keras.losses.sparse_categorical_crossentropy(
        labels, predictions)

    # Compute loss that is scaled by sample_weight and by global batch size.
    return tf.compute_average_loss(
        per_example_loss,
        sample_weight=sample_weight,
        global_batch_size=GLOBAL_BATCH_SIZE)
```

Args:

- **`per_example_loss`**: Per-example loss.
- **`sample_weight`**: Optional weighting for each example.
- **`global_batch_size`**: Optional global batch size value. Defaults to (size of first dimension of `losses`) * (number of replicas).

Returns:
Scalar loss value.

tf.nn.compute_average_loss

- Contents
- Aliases:
- Used in the tutorials:

Scales per-example losses with sample_weights and computes their average.

Aliases:

- `tf.compat.v1.nn.compute_average_loss`
- `tf.compat.v2.nn.compute_average_loss`
- `tf.nn.compute_average_loss`

```
tf.nn.compute_average_loss(
    per_example_loss,
    sample_weight=None,
    global_batch_size=None
)
```

Defined in `python/ops/nn_impl.py`.

Used in the tutorials:

- tf.distribute.Strategy with training loops

Usage with distribution strategy and custom training loop:

```
with strategy.scope():
  def compute_loss(labels, predictions, sample_weight=None):

    # If you are using a `Loss` class instead, set reduction to `NONE` so that
    # we can do the reduction afterwards and divide by global batch size.
    per_example_loss = tf.keras.losses.sparse_categorical_crossentropy(
        labels, predictions)

    # Compute loss that is scaled by sample_weight and by global batch size.
```

```
    return tf.compute_average_loss(
        per_example_loss,
        sample_weight=sample_weight,
        global_batch_size=GLOBAL_BATCH_SIZE)
```

Args:

- **`per_example_loss`**: Per-example loss.
- **`sample_weight`**: Optional weighting for each example.
- **`global_batch_size`**: Optional global batch size value. Defaults to (size of first dimension of `losses`) * (number of replicas).

Returns:
Scalar loss value.

tf.nn.conv1d_transpose

- Contents
- Aliases:

The transpose of `conv1d`.

Aliases:

- `tf.compat.v1.nn.conv1d_transpose`
- `tf.compat.v2.nn.conv1d_transpose`
- `tf.nn.conv1d_transpose`

```
tf.nn.conv1d_transpose(
    input,
    filters,
    output_shape,
    strides,
    padding='SAME',
    data_format='NWC',
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is really the transpose (gradient) of `conv1d` rather than an actual deconvolution.

Args:

- **`input`**: A 3-D `Tensor` of type `float` and shape `[batch, in_width, in_channels]` for `NWC`data format or `[batch, in_channels, in_width]` for `NCW` data format.
- **`filters`**: A 3-D `Tensor` with the same type as `value` and shape `[filter_width, output_channels, in_channels]`. `filter`'s `in_channels` dimension must match that of `value`.
- **`output_shape`**: A 1-D `Tensor`, containing three elements, representing the output shape of the deconvolution op.
- **`strides`**: An int or list of `ints` that has length `1` or `3`. The number of entries by which the filter is moved right at each step.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. `'NWC'` and `'NCW'` are supported.

- **`dilations`**: An int or list of `ints` that has length `1` or `3` which defaults to 1. The dilation factor for each dimension of input. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. Dilations in the batch and depth dimensions must be 1.
- **`name`**: Optional name for the returned tensor.

Returns:

A `Tensor` with the same type as `value`.

Raises:

- **`ValueError`**: If input/output depth does not match `filter`'s shape, if `output_shape` is not at 3-element vector, if `padding` is other than `'VALID'` or `'SAME'`, or if `data_format` is invalid.

tf.nn.conv2d

- Contents
- Aliases:

Computes a 2-D convolution given 4-D `input` and `filters` tensors.

Aliases:

- `tf.compat.v2.nn.conv2d`
- `tf.nn.conv2d`

```
tf.nn.conv2d(
    input,
    filters,
    strides,
    padding,
    data_format='NHWC',
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

In detail, with the default NHWC format,

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertices strides, `strides = [1, stride, stride, 1]`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. A 4-D tensor. The dimension order is interpreted according to the value of `data_format`, see below for details.

- **`filters`**: A `Tensor`. Must have the same type as `input`. A 4-D tensor of shape `[filter_height, filter_width, in_channels, out_channels]`
- **`strides`**: An int or list of `ints` that has length `1`, `2` or `4`. The stride of the sliding window for each dimension of `input`. If a single value is given it is replicated in the `H` and `W` dimension. By default the `N` and `C` dimensions are set to 1. The dimension order is determined by the value of `data_format`, see below for details.
- **`padding`**: Either the `string` `"SAME"` or `"VALID"` indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension. When explicit padding is used and data_format is `"NHWC"`, this should be in the form `[[0, 0], [pad_top, pad_bottom], [pad_left, pad_right], [0, 0]]`. When explicit padding used and data_format is `"NCHW"`, this should be in the form `[[0, 0], [0, 0], [pad_top, pad_bottom], [pad_left, pad_right]]`.
- **`data_format`**: An optional `string` from: `"NHWC", "NCHW"`. Defaults to `"NHWC"`. Specify the data format of the input and output data. With the default format "NHWC", the data is stored in the order of: [batch, height, width, channels]. Alternatively, the format could be "NCHW", the data storage order of: [batch, channels, height, width].
- **`dilations`**: An int or list of `ints` that has length `1`, `2` or `4`, defaults to 1. The dilation factor for each dimension of`input`. If a single value is given it is replicated in the `H` and `W` dimension. By default the `N` and `C` dimensions are set to 1. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions if a 4-d tensor must be 1.
- **`name`**: A name for the operation (optional).

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

tf.nn.conv2d_transpose

- Contents
- Aliases:

The transpose of `conv2d`.

Aliases:

- `tf.compat.v2.nn.conv2d_transpose`
- `tf.nn.conv2d_transpose`

```
tf.nn.conv2d_transpose(
    input,
    filters,
    output_shape,
    strides,
    padding='SAME',
    data_format='NHWC',
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is actually the transpose (gradient) of `conv2d` rather than an actual deconvolution.

Args:

- **`input`**: A 4-D `Tensor` of type `float` and shape `[batch, height, width, in_channels]` for `NHWC` data format or `[batch, in_channels, height, width]` for `NCHW` data format.

- **`filters`**: A 4-D `Tensor` with the same type as `input` and shape `[height, width, output_channels, in_channels]`. `filter`'s `in_channels` dimension must match that of `input`.
- **`output_shape`**: A 1-D `Tensor` representing the output shape of the deconvolution op.
- **`strides`**: An int or list of `ints` that has length `1`, `2` or `4`. The stride of the sliding window for each dimension of `input`. If a single value is given it is replicated in the `H` and `W` dimension. By default the `N` and `C` dimensions are set to 0. The dimension order is determined by the value of `data_format`, see below for details.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. 'NHWC' and 'NCHW' are supported.
- **`dilations`**: An int or list of `ints` that has length `1`, `2` or `4`, defaults to 1. The dilation factor for each dimension of`input`. If a single value is given it is replicated in the `H` and `W` dimension. By default the `N` and `C` dimensions are set to 1. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions if a 4-d tensor must be 1.
- **`name`**: Optional name for the returned tensor.

Returns:
A `Tensor` with the same type as `input`.

Raises:
- **`ValueError`**: If input/output depth does not match `filter`'s shape, or if padding is other than `'VALID'` or `'SAME'`.

tf.nn.conv3d

- Contents
- Aliases:

Computes a 3-D convolution given 5-D `input` and `filters` tensors.

Aliases:
- `tf.compat.v2.nn.conv3d`
- `tf.nn.conv3d`

```
tf.nn.conv3d(
    input,
    filters,
    strides,
    padding,
    data_format='NDHWC',
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
In signal processing, cross-correlation is a measure of similarity of two waveforms as a function of a time-lag applied to one of them. This is also known as a sliding dot product or sliding inner-product.
Our Conv3D implements a form of cross-correlation.

Args:
- **`input`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. Shape `[batch, in_depth, in_height, in_width, in_channels]`.

- **`filters`**: A `Tensor`. Must have the same type as `input`. Shape `[filter_depth, filter_height, filter_width, in_channels, out_channels]`. `in_channels` must match between `input` and `filters`.
- **`strides`**: A list of `ints` that has length `>= 5`. 1-D tensor of length 5. The stride of the sliding window for each dimension of `input`. Must have `strides[0] = strides[4] = 1`.
- **`padding`**: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- **`data_format`**: An optional `string` from: `"NDHWC", "NCDHW"`. Defaults to `"NDHWC"`. The data format of the input and output data. With the default format "NDHWC", the data is stored in the order of: [batch, in_depth, in_height, in_width, in_channels]. Alternatively, the format could be "NCDHW", the data storage order is: [batch, in_channels, in_depth, in_height, in_width].
- **`dilations`**: An optional list of `ints`. Defaults to `[1, 1, 1, 1, 1]`. 1-D tensor of length 5. The dilation factor for each dimension of `input`. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions must be 1.
- **`name`**: A name for the operation (optional).

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

tf.nn.conv3d_transpose

- Contents
- Aliases:

The transpose of `conv3d`.

Aliases:

- `tf.compat.v2.nn.conv3d_transpose`
- `tf.nn.conv3d_transpose`

```
tf.nn.conv3d_transpose(
    input,
    filters,
    output_shape,
    strides,
    padding='SAME',
    data_format='NDHWC',
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is actually the transpose (gradient) of `conv2d` rather than an actual deconvolution.

Args:

- **`input`**: A 5-D `Tensor` of type `float` and shape `[batch, height, width, in_channels]` for `NHWC` data format or `[batch, in_channels, height, width]` for `NCHW` data format.
- **`filters`**: A 5-D `Tensor` with the same type as `value` and shape `[height, width, output_channels, in_channels]`. `filter`'s `in_channels` dimension must match that of `value`.
- **`output_shape`**: A 1-D `Tensor` representing the output shape of the deconvolution op.
- **`strides`**: An int or list of `ints` that has length `1`, `3` or `5`. The stride of the sliding window for each dimension of `input`. If a single value is given it is replicated in the `D`, `H` and `W` dimension. By default

the `N` and `C` dimensions are set to 0. The dimension order is determined by the value of `data_format`, see below for details.

- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. 'NDHWC' and 'NCDHW' are supported.
- **`dilations`**: An int or list of `ints` that has length `1`, `3` or `5`, defaults to 1. The dilation factor for each dimension of`input`. If a single value is given it is replicated in the `D`, `H` and `W`dimension. By default the `N` and `C` dimensions are set to 1. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions if a 5-d tensor must be 1.
- **`name`**: Optional name for the returned tensor.

Returns:
A `Tensor` with the same type as `value`.

tf.nn.convolution

- Contents
- Aliases:

Computes sums of N-D convolutions (actually cross-correlation).

Aliases:

- `tf.compat.v2.nn.convolution`
- `tf.nn.convolution`

```
tf.nn.convolution(
    input,
    filters,
    strides=None,
    padding='VALID',
    data_format=None,
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This also supports either output striding via the optional `strides` parameter or atrous convolution (also known as convolution with holes or dilated convolution, based on the French word "trous" meaning holes in English) via the optional `dilations` parameter. Currently, however, output striding is not supported for atrous convolutions.
Specifically, in the case that `data_format` does not start with "NC", given a rank (N+2) `input` Tensor of shape
[num_batches, input_spatial_shape[0], ..., input_spatial_shape[N-1], num_input_channels],
a rank (N+2) `filters` Tensor of shape
[spatial_filter_shape[0], ..., spatial_filter_shape[N-1], num_input_channels, num_output_channels],
an optional `dilations` tensor of shape N specifying the filter upsampling/input downsampling rate, and an optional list of N `strides` (defaulting [1]*N), this computes for each N-D spatial output position (x[0], ..., x[N-1]):

```
  output[b, x[0], ..., x[N-1], k] =
      sum_{z[0], ..., z[N-1], q}
          filter[z[0], ..., z[N-1], q, k] *
          padded_input[b,
                       x[0]*strides[0] + dilation_rate[0]*z[0],
```

```
    ...,
    x[N-1]*strides[N-1] + dilation_rate[N-1]*z[N-1],
    q]
```

where b is the index into the batch, k is the output channel number, q is the input channel number, and z is the N-D spatial offset within the filter. Here, `padded_input` is obtained by zero padding the input using an effective spatial filter shape of `(spatial_filter_shape-1) * dilation_rate + 1` and output striding `strides` as described in the comment here.
In the case that `data_format` does start with `"NC"`, the `input` and output (but not the `filters`) are simply transposed as follows:
convolution(input, data_format, **kwargs) = tf.transpose(convolution(tf.transpose(input, [0] + range(2,N+2) + [1]), **kwargs), [0, N+1] + range(1, N+1))
It is required that 1 <= N <= 3.

Args:

- **`input`**: An (N+2)-D `Tensor` of type `T`, of shape `[batch_size] + input_spatial_shape + [in_channels]` if data_format does not start with "NC" (default), or `[batch_size, in_channels] + input_spatial_shape` if data_format starts with "NC".
- **`filters`**: An (N+2)-D `Tensor` with the same type as `input` and shape`spatial_filter_shape + [in_channels, out_channels]`.
- **`padding`**: A string, either `"VALID"` or `"SAME"`. The padding algorithm.
- **`strides`**: Optional. Sequence of N ints >= 1. Specifies the output stride. Defaults to [1]*N. If any value of strides is > 1, then all values of dilation_rate must be 1.
- **`dilations`**: Optional. Sequence of N ints >= 1. Specifies the filter upsampling/input downsampling rate. In the literature, the same parameter is sometimes called `input stride` or `dilation`. The effective filter size used for the convolution will be `spatial_filter_shape + (spatial_filter_shape - 1) * (rate - 1)`, obtained by inserting (dilation_rate[i]-1) zeros between consecutive elements of the original filter in each spatial dimension i. If any value of dilation_rate is > 1, then all values of strides must be 1.
- **`name`**: Optional name for the returned tensor.
- **`data_format`**: A string or None. Specifies whether the channel dimension of the `input` and output is the last dimension (default, or if `data_format` does not start with "NC"), or the second dimension (if `data_format` starts with "NC"). For N=1, the valid values are "NWC" (default) and "NCW". For N=2, the valid values are "NHWC" (default) and "NCHW". For N=3, the valid values are "NDHWC" (default) and "NCDHW".
- **`filters`**: Alias of filter.
- **`dilations`**: Alias of dilation_rate.

Returns:

A `Tensor` with the same type as `input` of shape

```
`[batch_size] + output_spatial_shape + [out_channels]`
```

if data_format is None or does not start with "NC", or

```
`[batch_size, out_channels] + output_spatial_shape`
```

if data_format starts with "NC", where `output_spatial_shape` depends on the value of `padding`.
If padding == "SAME": output_spatial_shape[i] = ceil(input_spatial_shape[i] / strides[i])
If padding == "VALID": output_spatial_shape[i] = ceil((input_spatial_shape[i] - (spatial_filter_shape[i]-1) * dilation_rate[i]) / strides[i]).

Raises:

- **`ValueError`**: If input/output depth does not match `filters` shape, if padding is other than `"VALID"` or `"SAME"`, or if data_format is invalid.

tf.nn.conv_transpose

- Contents
- Aliases:

The transpose of `convolution`.

Aliases:

- `tf.compat.v1.nn.conv_transpose`
- `tf.compat.v2.nn.conv_transpose`
- `tf.nn.conv_transpose`

```
tf.nn.conv_transpose(
    input,
    filters,
    output_shape,
    strides,
    padding='SAME',
    data_format=None,
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is actually the transpose (gradient) of `convolution` rather than an actual deconvolution.

Args:

- **`input`**: An N+2 dimensional `Tensor` of shape `[batch_size] + input_spatial_shape + [in_channels]` if data_format does not start with "NC" (default), or `[batch_size, in_channels] + input_spatial_shape` if data_format starts with "NC". It must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
- **`filters`**: An N+2 dimensional `Tensor` with the same type as `input` and shape `spatial_filter_shape + [in_channels, out_channels]`.
- **`output_shape`**: A 1-D `Tensor` representing the output shape of the deconvolution op.
- **`strides`**: An int or list of `ints` that has length `1`, `N` or `N+2`. The stride of the sliding window for each dimension of `input`. If a single value is given it is replicated in the spatial dimensions. By default the `N` and `C` dimensions are set to 0. The dimension order is determined by the value of `data_format`, see below for details.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string or None. Specifies whether the channel dimension of the `input` and output is the last dimension (default, or if `data_format` does not start with "NC"), or the second dimension (if `data_format` starts with "NC"). For N=1, the valid values are "NWC" (default) and "NCW". For N=2, the valid values are "NHWC" (default) and "NCHW". For N=3, the valid values are "NDHWC" (default) and "NCDHW".
- **`dilations`**: An int or list of `ints` that has length `1`, `N` or `N+2`, defaults to 1. The dilation factor for each dimension of`input`. If a single value is given it is replicated in the spatial dimensions. By default the `N` and `C` dimensions are set to 1. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details.
- **`name`**: A name for the operation (optional). If not specified "conv_transpose" is used.

Returns:
A `Tensor` with the same type as `value`.

tf.nn.crelu

- Contents
- Aliases:

Computes Concatenated ReLU.

Aliases:

- `tf.compat.v2.nn.crelu`
- `tf.nn.crelu`

```
tf.nn.crelu(
    features,
    axis=-1,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
Concatenates a ReLU which selects only the positive part of the activation with a ReLU which selects only the *negative* part of the activation. Note that as a result this non-linearity doubles the depth of the activations. Source: Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units. W. Shang, et al.

Args:

- **`features`**: A `Tensor` with type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.
- **`name`**: A name for the operation (optional).
- **`axis`**: The axis that the output values are concatenated along. Default is -1.

Returns:
A `Tensor` with the same type as `features`.

tf.nn.ctc_beam_search_decoder

- Contents
- Aliases:

Performs beam search decoding on the logits given in input.

Aliases:

- `tf.compat.v1.nn.ctc_beam_search_decoder_v2`
- `tf.compat.v2.nn.ctc_beam_search_decoder`
- `tf.nn.ctc_beam_search_decoder`

```
tf.nn.ctc_beam_search_decoder(
    inputs,
    sequence_length,
    beam_width=100,
    top_paths=1
)
```

Defined in `python/ops/ctc_ops.py`.
Note The `ctc_greedy_decoder` is a special case of the `ctc_beam_search_decoder` with `top_paths=1` and `beam_width=1` (but that decoder is faster for this special case).

Args:

- **`inputs`**: 3-D `float` `Tensor`, size `[max_time, batch_size, num_classes]`. The logits.
- **`sequence_length`**: 1-D `int32` vector containing sequence lengths, having size `[batch_size]`.
- **`beam_width`**: An int scalar >= 0 (beam search beam width).
- **`top_paths`**: An int scalar >= 0, <= beam_width (controls output size).

Returns:

A tuple `(decoded, log_probabilities)` where

- **`decoded`**: A list of length top_paths, where `decoded[j]` is a `SparseTensor` containing the decoded outputs:
  `decoded[j].indices`: Indices matrix `[total_decoded_outputs[j], 2]`; The rows store: `[batch, time]`.
  `decoded[j].values`: Values vector, size `[total_decoded_outputs[j]]`. The vector stores the decoded classes for beam `j`.
  `decoded[j].dense_shape`: Shape vector, size `(2)`. The shape values are: `[batch_size, max_decoded_length[j]]`.
- **`log_probability`**: A `float` matrix `[batch_size, top_paths]` containing sequence log-probabilities.

tf.nn.ctc_greedy_decoder

- Contents
- Aliases:

Performs greedy decoding on the logits given in input (best path).

Aliases:

- `tf.compat.v1.nn.ctc_greedy_decoder`
- `tf.compat.v2.nn.ctc_greedy_decoder`
- `tf.nn.ctc_greedy_decoder`

```
tf.nn.ctc_greedy_decoder(
    inputs,
    sequence_length,
    merge_repeated=True
)
```

Defined in `python/ops/ctc_ops.py`.

Note: Regardless of the value of merge_repeated, if the maximum index of a given time and batch corresponds to the blank index **`(num_classes - 1)`**, no new element is emitted.

If `merge_repeated` is `True`, merge repeated classes in output. This means that if consecutive logits' maximum indices are the same, only the first of these is emitted. The sequence `A B B * B * B`(where '*' is the blank label) becomes

- `A B B B` if `merge_repeated=True`.
- `A B B B B` if `merge_repeated=False`.

Args:

- **`inputs`**: 3-D `float` `Tensor` sized `[max_time, batch_size, num_classes]`. The logits.
- **`sequence_length`**: 1-D `int32` vector containing sequence lengths, having size `[batch_size]`.
- **`merge_repeated`**: Boolean. Default: True.

Returns:

A tuple `(decoded, neg_sum_logits)` where

- **`decoded`**: A single-element list. `decoded[0]` is an `SparseTensor` containing the decoded outputs s.t.:
  `decoded.indices`: Indices matrix `(total_decoded_outputs, 2)`. The rows store: `[batch, time]`.

`decoded.values`: Values vector, size `(total_decoded_outputs)`. The vector stores the decoded classes.
`decoded.dense_shape`: Shape vector, size `(2)`. The shape values are: `[batch_size, max_decoded_length]`

- **`neg_sum_logits`**: A `float` matrix `(batch_size x 1)` containing, for the sequence found, the negative of the sum of the greatest logit at each timeframe.

tf.nn.ctc_loss

- Contents
- Aliases:

Computes CTC (Connectionist Temporal Classification) loss.

Aliases:

- `tf.compat.v1.nn.ctc_loss_v2`
- `tf.compat.v2.nn.ctc_loss`
- `tf.nn.ctc_loss`

```
tf.nn.ctc_loss(
    labels,
    logits,
    label_length,
    logit_length,
    logits_time_major=True,
    unique=None,
    blank_index=None,
    name=None
)
```

Defined in `python/ops/ctc_ops.py`.
This op implements the CTC loss as presented in the article:
[A. Graves, S. Fernandez, F. Gomez, J. Schmidhuber. Connectionist Temporal Classification: Labeling Unsegmented Sequence Data with Recurrent Neural Networks. ICML 2006, Pittsburgh, USA, pp. 369-376.]

Notes:

- Same as the "Classic CTC" in TensorFlow 1.x's tf.compat.v1.nn.ctc_loss setting of preprocess_collapse_repeated=False, ctc_merge_repeated=True
- Labels may be supplied as either a dense, zero-padded tensor with a vector of label sequence lengths OR as a SparseTensor.
- On TPU and GPU:
- Only dense padded labels are supported.
- On CPU:
- Caller may use SparseTensor or dense padded labels but calling with a SparseTensor will be significantly faster.
- Default blank label is 0 rather num_classes - 1, unless overridden by blank_index.

Args:

- **`labels`**: tensor of shape [batch_size, max_label_seq_length] or SparseTensor
- **`logits`**: tensor of shape [frames, batch_size, num_labels], if logits_time_major == False, shape is [batch_size, frames, num_labels].
- **`label_length`**: tensor of shape [batch_size], None if labels is SparseTensor Length of reference label sequence in labels.
- **`logit_length`**: tensor of shape [batch_size] Length of input sequence in logits.

- `logits_time_major`: (optional) If True (default), logits is shaped [time, batch, logits]. If False, shape is [batch, time, logits]
- `unique`: (optional) Unique label indices as computed by ctc_unique_labels(labels). If supplied, enable a faster, memory efficient implementation on TPU.
- `blank_index`: (optional) Set the class index to use for the blank label. Negative values will start from num_classes, ie, -1 will reproduce the ctc_loss behavior of using num_classes - 1 for the blank symbol. There is some memory/performance overhead to switching from the default of 0 as an additional shifted copy of the logits may be created.
- `name`: A name for this `Op`. Defaults to "ctc_loss_dense".

Returns:

- `loss`: tensor of shape [batch_size], negative log probabilities.

tf.nn.ctc_unique_labels

- Contents
- Aliases:

Get unique labels and indices for batched labels for `tf.nn.ctc_loss`.

Aliases:

- `tf.compat.v1.nn.ctc_unique_labels`
- `tf.compat.v2.nn.ctc_unique_labels`
- `tf.nn.ctc_unique_labels`

```
tf.nn.ctc_unique_labels(
    labels,
    name=None
)
```

Defined in `python/ops/ctc_ops.py`.

For use with `tf.nn.ctc_loss` optional argument `unique`: This op can be used to preprocess labels in input pipeline to for better speed/memory use computing the ctc loss on TPU.

Example:

ctc_unique_labels([[3, 4, 4, 3]]) -> unique labels padded with 0: [[3, 4, 0, 0]] indices of original labels in unique: [0, 1, 1, 0]

Args:

- `labels`: tensor of shape [batch_size, max_label_length] padded with 0.
- `name`: A name for this `Op`. Defaults to "ctc_unique_labels".

Returns:

tuple of - unique labels, tensor of shape `[batch_size, max_label_length]` - indices into unique labels, shape `[batch_size, max_label_length]`

tf.nn.depthwise_conv2d

- Contents
- Aliases:

Depthwise 2-D convolution.

Aliases:

- `tf.compat.v2.nn.depthwise_conv2d`
- `tf.nn.depthwise_conv2d`

```
tf.nn.depthwise_conv2d(
    input,
    filter,
```

```
    strides,
    padding,
    data_format=None,
    dilations=None,
    name=None
)
```

Defined in python/ops/nn_impl.py.
Given a 4D input tensor ('NHWC' or 'NCHW' data formats) and a filter tensor of shape `[filter_height, filter_width, in_channels, channel_multiplier]` containing `in_channels` convolutional filters of depth 1, `depthwise_conv2d` applies a different filter to each input channel (expanding from 1 channel to `channel_multiplier` channels for each), then concatenates the results together. The output has `in_channels * channel_multiplier` channels.
In detail, with the default NHWC format,

```
output[b, i, j, k * channel_multiplier + q] = sum_{di, dj}
     filter[di, dj, k, q] * input[b, strides[1] * i + rate[0] * di,
                                     strides[2] * j + rate[1] * dj, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`. If any value in `rate` is greater than 1, we perform atrous depthwise convolution, in which case all values in the `strides` tensor must be equal to 1.

Args:

- **`input`**: 4-D with shape according to `data_format`.
- **`filter`**: 4-D with shape `[filter_height, filter_width, in_channels, channel_multiplier]`.
- **`strides`**: 1-D of size 4. The stride of the sliding window for each dimension of `input`.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: The data format for input. Either "NHWC" (default) or "NCHW".
- **`dilations`**: 1-D of size 2. The dilation rate in which we sample input values across the `height`and `width` dimensions in atrous convolution. If it is greater than 1, then all values of strides must be 1.
- **`name`**: A name for this operation (optional).

Returns:
A 4-D `Tensor` with shape according to `data_format`. E.g., for "NHWC" format, shape is `[batch, out_height, out_width, in_channels * channel_multiplier].`

tf.nn.depthwise_conv2d_backprop_filter

- Contents
- Aliases:

Computes the gradients of depthwise convolution with respect to the filter.

Aliases:

- `tf.compat.v1.nn.depthwise_conv2d_backprop_filter`
- `tf.compat.v1.nn.depthwise_conv2d_native_backprop_filter`
- `tf.compat.v2.nn.depthwise_conv2d_backprop_filter`
- `tf.nn.depthwise_conv2d_backprop_filter`

```
tf.nn.depthwise_conv2d_backprop_filter(
    input,
    filter_sizes,
    out_backprop,
    strides,
    padding,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. 4-D with shape based on `data_format`. For example, if `data_format` is 'NHWC' then `input` is a 4-D `[batch, in_height, in_width, in_channels]` tensor.
- **`filter_sizes`**: A `Tensor` of type `int32`. An integer vector representing the tensor shape of `filter`, where `filter` is a 4-D `[filter_height, filter_width, in_channels, depthwise_multiplier]` tensor.
- **`out_backprop`**: A `Tensor`. Must have the same type as `input`. 4-D with shape based on `data_format`. For example, if `data_format` is 'NHWC' then out_backprop shape is `[batch, out_height, out_width, out_channels]`. Gradients w.r.t. the output of the convolution.
- **`strides`**: A list of `ints`. The stride of the sliding window for each dimension of the input of the convolution.
- **`padding`**: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- **`data_format`**: An optional `string` from: `"NHWC", "NCHW"`. Defaults to `"NHWC"`. Specify the data format of the input and output data. With the default format "NHWC", the data is stored in the order of: [batch, height, width, channels]. Alternatively, the format could be "NCHW", the data storage order of: [batch, channels, height, width].
- **`dilations`**: An optional list of `ints`. Defaults to `[1, 1, 1, 1]`. 1-D tensor of length 4. The dilation factor for each dimension of `input`. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions must be 1.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.depthwise_conv2d_backprop_input

- Contents
- Aliases:

Computes the gradients of depthwise convolution with respect to the input.

Aliases:

- `tf.compat.v1.nn.depthwise_conv2d_backprop_input`
- `tf.compat.v1.nn.depthwise_conv2d_native_backprop_input`
- `tf.compat.v2.nn.depthwise_conv2d_backprop_input`
- `tf.nn.depthwise_conv2d_backprop_input`

```
tf.nn.depthwise_conv2d_backprop_input(
    input_sizes,
```

```
    filter,
    out_backprop,
    strides,
    padding,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Args:

- **`input_sizes`**: A `Tensor` of type `int32`. An integer vector representing the shape of `input`, based on `data_format`. For example, if `data_format` is 'NHWC' then `input` is a 4-D `[batch, height, width, channels]` tensor.
- **`filter`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. 4-D with shape `[filter_height, filter_width, in_channels, depthwise_multiplier]`.
- **`out_backprop`**: A `Tensor`. Must have the same type as `filter`. 4-D with shape based on `data_format`. For example, if `data_format` is 'NHWC' then out_backprop shape is `[batch, out_height, out_width, out_channels]`. Gradients w.r.t. the output of the convolution.
- **`strides`**: A list of `ints`. The stride of the sliding window for each dimension of the input of the convolution.
- **`padding`**: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- **`data_format`**: An optional `string` from: `"NHWC", "NCHW"`. Defaults to `"NHWC"`. Specify the data format of the input and output data. With the default format "NHWC", the data is stored in the order of: [batch, height, width, channels]. Alternatively, the format could be "NCHW", the data storage order of: [batch, channels, height, width].
- **`dilations`**: An optional list of `ints`. Defaults to `[1, 1, 1, 1]`. 1-D tensor of length 4. The dilation factor for each dimension of `input`. If set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`, see above for details. Dilations in the batch and depth dimensions must be 1.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.depth_to_space

- Contents
- Aliases:

DepthToSpace for tensors of type T.

Aliases:

- `tf.compat.v2.nn.depth_to_space`
- `tf.nn.depth_to_space`

```
tf.nn.depth_to_space(
    input,
    block_size,
    data_format='NHWC',
    name=None
)
```

Defined in `python/ops/array_ops.py`.
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 Y, X coordinates within each block of the output image are determined by the high order component of the input channel index.
- The depth of the input tensor must be divisible by `block_size * block_size`.

The `data_format` attr specifies the layout of the input and output tensors with the following options: "NHWC": `[ batch, height, width, channels ]` "NCHW": `[ batch, channels, height, width ]` "NCHW_VECT_C": `qint8 [ batch, channels / 4, height, width, 4 ]`
It is useful to consider the operation as transforming a 6-D Tensor. e.g. for data_format = NHWC, Each element in the input tensor can be specified via 6 coordinates, ordered by decreasing memory layout significance as: n,iY,iX,bY,bX,oC (where n=batch index, iX, iY means X or Y coordinates within the input image, bX, bY means coordinates within the output block, oC means output channels). The output would be the input transposed to the following layout: n,iY,bY,iX,bX,oC
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 an input of shape `[1, 1, 1, 4]`, data_format = "NHWC" and block_size = 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.
- **`data_format`**: An optional `string` from: `"NHWC", "NCHW", "NCHW_VECT_C"`. Defaults to `"NHWC"`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.dilation2d

- Contents
- Aliases:

Computes the grayscale dilation of 4-D `input` and 3-D `filters` tensors.

Aliases:

- `tf.compat.v2.nn.dilation2d`
- `tf.nn.dilation2d`

```
tf.nn.dilation2d(
    input,
    filters,
    strides,
    padding,
    data_format,
    dilations,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

The `input` tensor has shape `[batch, in_height, in_width, depth]` and the `filters` tensor has shape `[filter_height, filter_width, depth]`, i.e., each input channel is processed independently of the others with its own structuring function. The `output` tensor has shape `[batch, out_height, out_width, depth]`. The spatial dimensions of the output tensor depend on the `padding` algorithm. We currently only support the default "NHWC" `data_format`.

In detail, the grayscale morphological 2-D dilation is the max-sum correlation (for consistency with `conv2d`, we use unmirrored filters):

```
output[b, y, x, c] =
   max_{dy, dx} input[b,
                      strides[1] * y + rates[1] * dy,
                      strides[2] * x + rates[2] * dx,
                      c] +
                filters[dy, dx, c]
```

Max-pooling is a special case when the filter has size equal to the pooling kernel size and contains all zeros.

Note on duality: The dilation of `input` by the `filters` is equal to the negation of the erosion of `-input` by the reflected `filters`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`. 4-D with shape `[batch, in_height, in_width, depth]`.
- `filters`: A `Tensor`. Must have the same type as `input`. 3-D with shape `[filter_height, filter_width, depth]`.
- `strides`: A list of `ints` that has length `>= 4`. The stride of the sliding window for each dimension of the input tensor. Must be: `[1, stride_height, stride_width, 1]`.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `data_format`: A `string`, only `"NCHW"` is currently supported.
- `dilations`: A list of `ints` that has length `>= 4`. The input stride for atrous morphological dilation. Must be: `[1, rate_height, rate_width, 1]`.
- `name`: A name for the operation (optional).

Returns:

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

tf.nn.dropout

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes dropout.

Aliases:

- `tf.compat.v2.nn.dropout`
- `tf.nn.dropout`

```
tf.nn.dropout(
    x,
    rate,
    noise_shape=None,
    seed=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Used in the guide:

- Writing layers and models with TensorFlow Keras

Used in the tutorials:

- tf.function

With probability `rate`, drops elements of `x`. Input that are kept are scaled up by `1 / (1 - rate)`, otherwise outputs `0`. The scaling is so that the expected sum is unchanged.

Note: The behavior of dropout has changed between TensorFlow 1.x and 2.x. When converting 1.x code, please use named arguments to ensure behavior stays consistent.

By default, each element is kept or dropped independently. If `noise_shape` is specified, it must bebroadcastable to the shape of `x`, and only dimensions with `noise_shape[i] == shape(x)[i]` will make independent decisions. For example, if `shape(x) = [k, l, m, n]` and `noise_shape = [k, 1, 1, n]`, each batch and channel component will be kept independently and each row and column will be kept or not kept together.

Args:

- **`x`**: A floating point tensor.
- **`rate`**: A scalar `Tensor` with the same type as x. The probability that each element is dropped. For example, setting rate=0.1 would drop 10% of input elements.
- **`noise_shape`**: A 1-D `Tensor` of type `int32`, representing the shape for randomly generated keep/drop flags.
- **`seed`**: A Python integer. Used to create random seeds. See `tf.compat.v1.set_random_seed`for behavior.
- **`name`**: A name for this operation (optional).

Returns:

A Tensor of the same shape of `x`.

Raises:

- **`ValueError`**: If `rate` is not in `(0, 1]` or if `x` is not a floating point tensor.

tf.nn.elu

- Contents
- Aliases:

Computes exponential linear: `exp(features) - 1` if < 0, `features` otherwise.

Aliases:

- `tf.compat.v1.nn.elu`
- `tf.compat.v2.nn.elu`
- `tf.nn.elu`

```
tf.nn.elu(
    features,
    name=None
)

```

Defined in generated file: `python/ops/gen_nn_ops.py`.

See [Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)]()

Args:

- **`features`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.embedding_lookup

- Contents
- Aliases:

Looks up `ids` in a list of embedding tensors.

Aliases:

- `tf.compat.v2.nn.embedding_lookup`
- `tf.nn.embedding_lookup`

```
tf.nn.embedding_lookup(
    params,
    ids,
    max_norm=None,
    name=None
```

```
)
```

Defined in `python/ops/embedding_ops.py`.

This function is used to perform parallel lookups on the list of tensors in `params`. It is a generalization of `tf.gather`, where `params` is interpreted as a partitioning of a large embedding tensor. `params`may be a `PartitionedVariable` as returned by using `tf.compat.v1.get_variable()` with a partitioner.

If `len(params) > 1`, each element `id` of `ids` is partitioned between the elements of `params`according to the `partition_strategy`. In all strategies, if the id space does not evenly divide the number of partitions, each of the first `(max_id + 1) % len(params)` partitions will be assigned one more id.

The `partition_strategy` is always `"div"` currently. This means that we assign ids to partitions in a contiguous manner. For instance, 13 ids are split across 5 partitions as: `[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10], [11, 12]]`

The results of the lookup are concatenated into a dense tensor. The returned tensor has shape `shape(ids) + shape(params)[1:]`.

Args:

- **`params`**: A single tensor representing the complete embedding tensor, or a list of P tensors all of same shape except for the first dimension, representing sharded embedding tensors. Alternatively, a `PartitionedVariable`, created by partitioning along dimension 0. Each element must be appropriately sized for the 'div' `partition_strategy`.
- **`ids`**: A `Tensor` with type `int32` or `int64` containing the ids to be looked up in `params`.
- **`max_norm`**: If not `None`, each embedding is clipped if its l2-norm is larger than this value.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` with the same type as the tensors in `params`.

Raises:

- **`ValueError`**: If `params` is empty.

tf.nn.embedding_lookup_sparse

- Contents
- Aliases:

Computes embeddings for the given ids and weights.

Aliases:

- `tf.compat.v2.nn.embedding_lookup_sparse`
- `tf.nn.embedding_lookup_sparse`

```
tf.nn.embedding_lookup_sparse(
    params,
    sp_ids,
    sp_weights,
    combiner=None,
    max_norm=None,
    name=None
)
```

Defined in `python/ops/embedding_ops.py`.

This op assumes that there is at least one id for each row in the dense tensor represented by sp_ids (i.e. there are no rows with empty features), and that all the indices of sp_ids are in canonical row-major order.

It also assumes that all id values lie in the range [0, p0), where p0 is the sum of the size of params along dimension 0.

Args:

- **`params`**: A single tensor representing the complete embedding tensor, or a list of P tensors all of same shape except for the first dimension, representing sharded embedding tensors. Alternatively, a `PartitionedVariable`, created by partitioning along dimension 0. Each element must be appropriately sized for `"div"` `partition_strategy`.
- **`sp_ids`**: N x M `SparseTensor` of int64 ids where N is typically batch size and M is arbitrary.
- **`sp_weights`**: either a `SparseTensor` of float / double weights, or `None` to indicate all weights should be taken to be 1. If specified, `sp_weights` must have exactly the same shape and indices as `sp_ids`.
- **`combiner`**: A string specifying the reduction op. Currently "mean", "sqrtn" and "sum" are supported. "sum" computes the weighted sum of the embedding results for each row. "mean" is the weighted sum divided by the total weight. "sqrtn" is the weighted sum divided by the square root of the sum of the squares of the weights.
- **`max_norm`**: If not `None`, each embedding is clipped if its l2-norm is larger than this value, before combining.
- **`name`**: Optional name for the op.

Returns:

A dense tensor representing the combined embeddings for the sparse ids. For each row in the dense tensor represented by `sp_ids`, the op looks up the embeddings for all ids in that row, multiplies them by the corresponding weight, and combines these embeddings as specified.
In other words, if
`shape(combined params) = [p0, p1, ..., pm]`
and
`shape(sp_ids) = shape(sp_weights) = [d0, d1, ..., dn]`
then
`shape(output) = [d0, d1, ..., dn-1, p1, ..., pm]`.
For instance, if params is a 10x20 matrix, and sp_ids / sp_weights are

```
[0, 0]: id 1, weight 2.0
[0, 1]: id 3, weight 0.5
[1, 0]: id 0, weight 1.0
[2, 3]: id 1, weight 3.0
```

with `combiner`="mean", then the output will be a 3x20 matrix where

```
output[0, :] = (params[1, :] * 2.0 + params[3, :] * 0.5) / (2.0 + 0.5)
output[1, :] = (params[0, :] * 1.0) / 1.0
output[2, :] = (params[1, :] * 3.0) / 3.0
```

Raises:

- **`TypeError`**: If `sp_ids` is not a `SparseTensor`, or if `sp_weights` is neither `None` nor `SparseTensor`.
- **`ValueError`**: If `combiner` is not one of {"mean", "sqrtn", "sum"}.

tf.nn.erosion2d

- Contents
- Aliases:

Computes the grayscale erosion of 4-D `value` and 3-D `filters` tensors.

Aliases:

- `tf.compat.v2.nn.erosion2d`

- `tf.nn.erosion2d`

```
tf.nn.erosion2d(
    value,
    filters,
    strides,
    padding,
    data_format,
    dilations,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
The `value` tensor has shape `[batch, in_height, in_width, depth]` and the `filters` tensor has shape `[filters_height, filters_width, depth]`, i.e., each input channel is processed independently of the others with its own structuring function. The `output` tensor has shape `[batch, out_height, out_width, depth]`. The spatial dimensions of the output tensor depend on the `padding` algorithm. We currently only support the default "NHWC" `data_format`.
In detail, the grayscale morphological 2-D erosion is given by:

```
output[b, y, x, c] =
   min_{dy, dx} value[b,
                      strides[1] * y - dilations[1] * dy,
                      strides[2] * x - dilations[2] * dx,
                      c] -
                filters[dy, dx, c]
```

Duality: The erosion of `value` by the `filters` is equal to the negation of the dilation of `-value` by the reflected `filters`.

Args:

- **`value`**: A `Tensor`. 4-D with shape `[batch, in_height, in_width, depth]`.
- **`filters`**: A `Tensor`. Must have the same type as `value`. 3-D with shape `[filters_height, filters_width, depth]`.
- **`strides`**: A list of `ints` that has length `>= 4`. 1-D of length 4. The stride of the sliding window for each dimension of the input tensor. Must be: `[1, stride_height, stride_width, 1]`.
- **`padding`**: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- **`data_format`**: A `string`, only `"NHWC"` is currently supported.
- **`dilations`**: A list of `ints` that has length `>= 4`. 1-D of length 4. The input stride for atrous morphological dilation. Must be: `[1, rate_height, rate_width, 1]`.
- **`name`**: A name for the operation (optional). If not specified "erosion2d" is used.

Returns:

A `Tensor`. Has the same type as `value`. 4-D with shape `[batch, out_height, out_width, depth]`.

Raises:

- **`ValueError`**: If the `value` depth does not match `filters`' shape, or if padding is other than `'VALID'` or `'SAME'`.

tf.nn.fractional_avg_pool

- Contents
- Aliases:

Performs fractional average pooling on the input.

Aliases:

- `tf.compat.v2.nn.fractional_avg_pool`
- `tf.nn.fractional_avg_pool`

```
tf.nn.fractional_avg_pool(
    value,
    pooling_ratio,
    pseudo_random=False,
    overlapping=False,
    seed=0,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
Fractional average pooling is similar to Fractional max pooling in the pooling region generation step. The only difference is that after pooling regions are generated, a mean operation is performed instead of a max operation in each pooling region.

Args:

- **`value`**: A `Tensor`. 4-D with shape `[batch, height, width, channels]`.
- **`pooling_ratio`**: A list of `floats` that has length >= 4. Pooling ratio for each dimension of `value`, currently only supports row and col dimension and should be >= 1.0. For example, a valid pooling ratio looks like [1.0, 1.44, 1.73, 1.0]. The first and last elements must be 1.0 because we don't allow pooling on batch and channels dimensions. 1.44 and 1.73 are pooling ratio on height and width dimensions respectively.
- **`pseudo_random`**: An optional `bool`. Defaults to `False`. When set to `True`, generates the pooling sequence in a pseudorandom fashion, otherwise, in a random fashion. Check paper [Benjamin Graham, Fractional Max-Pooling]() for difference between pseudorandom and random.
- **`overlapping`**: An optional `bool`. Defaults to `False`. When set to `True`, it means when pooling, the values at the boundary of adjacent pooling cells are used by both cells. For example:`index 0 1 2 3 4` `value 20 5 16 3 7` If the pooling sequence is [0, 2, 4], then 16, at index 2 will be used twice. The result would be [20, 16] for fractional avg pooling.
- **`seed`**: An optional `int`. Defaults to `0`. If set to be non-zero, the random number generator is seeded by the given seed. Otherwise it is seeded by a random seed.
- **`name`**: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (`output`, `row_pooling_sequence`, `col_pooling_sequence`). output: Output `Tensor` after fractional avg pooling. Has the same type as `value`. row_pooling_sequence: A `Tensor` of type `int64`. col_pooling_sequence: A `Tensor` of type `int64`.

tf.nn.fractional_max_pool

- Contents
- Aliases:

Performs fractional max pooling on the input.

Aliases:

- `tf.compat.v2.nn.fractional_max_pool`
- `tf.nn.fractional_max_pool`

```
tf.nn.fractional_max_pool(
    value,
    pooling_ratio,
```

```
    pseudo_random=False,
    overlapping=False,
    seed=0,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Fractional max pooling is slightly different than regular max pooling. In regular max pooling, you downsize an input set by taking the maximum value of smaller N x N subsections of the set (often 2x2), and try to reduce the set by a factor of N, where N is an integer. Fractional max pooling, as you might expect from the word "fractional", means that the overall reduction ratio N does not have to be an integer.

The sizes of the pooling regions are generated randomly but are fairly uniform. For example, let's look at the height dimension, and the constraints on the list of rows that will be pool boundaries.

First we define the following:

1. input_row_length : the number of rows from the input set
2. output_row_length : which will be smaller than the input
3. alpha = input_row_length / output_row_length : our reduction ratio
4. K = floor(alpha)
5. row_pooling_sequence : this is the result list of pool boundary rows

Then, row_pooling_sequence should satisfy:

1. a[0] = 0 : the first value of the sequence is 0
2. a[end] = input_row_length : the last value of the sequence is the size
3. K <= (a[i+1] - a[i]) <= K+1 : all intervals are K or K+1 size
4. length(row_pooling_sequence) = output_row_length+1

For more details on fractional max pooling, see this paper: Benjamin Graham, Fractional Max-Pooling

Args:

- **`value`**: A `Tensor`. 4-D with shape `[batch, height, width, channels]`.
- **`pooling_ratio`**: An int or list of `ints` that has length `1`, `2` or `4`. Pooling ratio for each dimension of `value`, currently only supports row and col dimension and should be >= 1.0. For example, a valid pooling ratio looks like [1.0, 1.44, 1.73, 1.0]. The first and last elements must be 1.0 because we don't allow pooling on batch and channels dimensions. 1.44 and 1.73 are pooling ratio on height and width dimensions respectively.
- **`pseudo_random`**: An optional `bool`. Defaults to `False`. When set to `True`, generates the pooling sequence in a pseudorandom fashion, otherwise, in a random fashion. Check paper Benjamin Graham, Fractional Max-Pooling for difference between pseudorandom and random.
- **`overlapping`**: An optional `bool`. Defaults to `False`. When set to `True`, it means when pooling, the values at the boundary of adjacent pooling cells are used by both cells. For example:`index 0 1 2 3 4 value 20 5 16 3 7` If the pooling sequence is [0, 2, 4], then 16, at index 2 will be used twice. The result would be [20, 16] for fractional max pooling.
- **`seed`**: An optional `int`. Defaults to `0`. If set to be non-zero, the random number generator is seeded by the given seed. Otherwise it is seeded by a random seed.
- **`name`**: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (`output`, `row_pooling_sequence`, `col_pooling_sequence`). output: Output `Tensor` after fractional max pooling. Has the same type as `value`. row_pooling_sequence: A `Tensor` of type `int64`. col_pooling_sequence: A `Tensor` of type `int64`.

tf.nn.l2_loss

- Contents

- Aliases:

L2 Loss.

Aliases:

- `tf.compat.v1.nn.l2_loss`
- `tf.compat.v2.nn.l2_loss`
- `tf.nn.l2_loss`

```
tf.nn.l2_loss(
    t,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Computes half the L2 norm of a tensor without the `sqrt`:

```
output = sum(t ** 2) / 2
```

Args:

- **`t`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`. Typically 2-D, but may have any dimensions.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.leaky_relu

- Contents
- Aliases:

Compute the Leaky ReLU activation function.

Aliases:

- `tf.compat.v1.nn.leaky_relu`
- `tf.compat.v2.nn.leaky_relu`
- `tf.nn.leaky_relu`

```
tf.nn.leaky_relu(
    features,
    alpha=0.2,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Source: [Rectifier Nonlinearities Improve Neural Network Acoustic Models. AL Maas, AY Hannun, AY Ng - Proc. ICML, 2013]().

Args:

- **`features`**: A `Tensor` representing preactivation values. Must be one of the following types: `float16`, `float32`, `float64`, `int32`, `int64`.
- **`alpha`**: Slope of the activation function at x < 0.
- **`name`**: A name for the operation (optional).

Returns:

The activation value.

tf.nn.local_response_normalization

- Contents
- Aliases:

Local Response Normalization.

Aliases:

- `tf.compat.v1.nn.local_response_normalization`
- `tf.compat.v1.nn.lrn`
- `tf.compat.v2.nn.local_response_normalization`
- `tf.compat.v2.nn.lrn`
- `tf.nn.local_response_normalization`
- `tf.nn.lrn`

```
tf.nn.local_response_normalization(
    input,
    depth_radius=5,
    bias=1,
    alpha=1,
    beta=0.5,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

The 4-D `input` tensor is treated as a 3-D array of 1-D vectors (along the last dimension), and each vector is normalized independently. Within a given vector, each component is divided by the weighted, squared sum of inputs within `depth_radius`. In detail,

```
sqr_sum[a, b, c, d] =
    sum(input[a, b, c, d - depth_radius : d + depth_radius + 1] ** 2)
output = input / (bias + alpha * sqr_sum) ** beta
```

For details, see [Krizhevsky et al., ImageNet classification with deep convolutional neural networks (NIPS 2012)].

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`. 4-D.
- **`depth_radius`**: An optional `int`. Defaults to `5`. 0-D. Half-width of the 1-D normalization window.
- **`bias`**: An optional `float`. Defaults to `1`. An offset (usually positive to avoid dividing by 0).
- **`alpha`**: An optional `float`. Defaults to `1`. A scale factor, usually positive.
- **`beta`**: An optional `float`. Defaults to `0.5`. An exponent.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.log_poisson_loss

- Contents
- Aliases:

Computes log Poisson loss given `log_input`.

Aliases:

- `tf.compat.v1.nn.log_poisson_loss`
- `tf.compat.v2.nn.log_poisson_loss`

- `tf.nn.log_poisson_loss`

```
tf.nn.log_poisson_loss(
    targets,
    log_input,
    compute_full_loss=False,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

Gives the log-likelihood loss between the prediction and the target under the assumption that the target has a Poisson distribution. Caveat: By default, this is not the exact loss, but the loss minus a constant term [log(z!)]. That has no effect for optimization, but does not play well with relative loss comparisons. To compute an approximation of the log factorial term, specify compute_full_loss=True to enable Stirling's Approximation.

For brevity, let `c = log(x) = log_input`, `z = targets`. The log Poisson loss is

```
  -log(exp(-x) * (x^z) / z!)
= -log(exp(-x) * (x^z)) + log(z!)
~ -log(exp(-x)) - log(x^z) [+ z * log(z) - z + 0.5 * log(2 * pi * z)]
    [ Note the second term is the Stirling's Approximation for log(z!).
      It is invariant to x and does not affect optimization, though
      important for correct relative loss comparisons. It is only
      computed when compute_full_loss == True. ]
= x - z * log(x) [+ z * log(z) - z + 0.5 * log(2 * pi * z)]
= exp(c) - z * c [+ z * log(z) - z + 0.5 * log(2 * pi * z)]
```

Args:

- **`targets`**: A `Tensor` of the same type and shape as `log_input`.
- **`log_input`**: A `Tensor` of type `float32` or `float64`.
- **`compute_full_loss`**: whether to compute the full loss. If false, a constant term is dropped in favor of more efficient optimization.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `log_input` with the componentwise logistic losses.

Raises:

- **`ValueError`**: If `log_input` and `targets` do not have the same shape.

tf.nn.log_softmax

- Contents
- Aliases:

Computes log softmax activations.

Aliases:

- `tf.compat.v2.math.log_softmax`
- `tf.compat.v2.nn.log_softmax`
- `tf.math.log_softmax`
- `tf.nn.log_softmax`

```
tf.nn.log_softmax(
    logits,
    axis=None,
```

```
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

For each batch `i` and class `j` we have

```
logsoftmax = logits - log(reduce_sum(exp(logits), axis))
```

Args:

- **`logits`**: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- **`axis`**: The dimension softmax would be performed on. The default is -1 which indicates the last dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `logits`. Same shape as `logits`.

Raises:

- **`InvalidArgumentError`**: if `logits` is empty or `axis` is beyond the last dimension of `logits`.

tf.nn.max_pool

- Contents
- Aliases:

Performs the max pooling on the input.

Aliases:

- `tf.compat.v1.nn.max_pool_v2`
- `tf.compat.v2.nn.max_pool`
- `tf.nn.max_pool`

```
tf.nn.max_pool(
    input,
    ksize,
    strides,
    padding,
    data_format=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Args:

- **`input`**: Tensor of rank N+2, of shape `[batch_size] + input_spatial_shape + [num_channels]` if `data_format` does not start with "NC" (default), or `[batch_size, num_channels] + input_spatial_shape` if data_format starts with "NC". Pooling happens over the spatial dimensions only.
- **`ksize`**: An int or list of `ints` that has length `1`, `N` or `N+2`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `N` or `N+2`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.

- **`data_format`**: A string. Specifies the channel dimension. For N=1 it can be either "NWC" (default) or "NCW", for N=2 it can be either "NHWC" (default) or "NCHW" and for N=3 either "NDHWC" (default) or "NCDHW".
- **`name`**: Optional name for the operation.

Returns:

A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.max_pool1d

- Contents
- Aliases:

Performs the max pooling on the input.

Aliases:

- `tf.compat.v1.nn.max_pool1d`
- `tf.compat.v2.nn.max_pool1d`
- `tf.nn.max_pool1d`

```
tf.nn.max_pool1d(
    input,
    ksize,
    strides,
    padding,
    data_format='NWC',
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Note internally this op reshapes and uses the underlying 2d operation.

Args:

- **`input`**: A 3-D `Tensor` of the format specified by `data_format`.
- **`ksize`**: An int or list of `ints` that has length `1` or `3`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1` or `3`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: An optional string from: "NWC", "NCW". Defaults to "NWC".
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.max_pool2d

- Contents
- Aliases:

Performs the max pooling on the input.

Aliases:

- `tf.compat.v1.nn.max_pool2d`
- `tf.compat.v2.nn.max_pool2d`
- `tf.nn.max_pool2d`

```
tf.nn.max_pool2d(
    input,
    ksize,
    strides,
    padding,
    data_format='NHWC',
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Args:

- **`input`**: A 4-D `Tensor` of the format specified by `data_format`.
- **`ksize`**: An int or list of `ints` that has length `1`, `2` or `4`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `2` or `4`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: A string. 'NHWC', 'NCHW' and 'NCHW_VECT_C' are supported.
- **`name`**: Optional name for the operation.

Returns:

A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.max_pool3d

- Contents
- Aliases:

Performs the max pooling on the input.

Aliases:

- `tf.compat.v1.nn.max_pool3d`
- `tf.compat.v2.nn.max_pool3d`
- `tf.nn.max_pool3d`

```
tf.nn.max_pool3d(
    input,
    ksize,
    strides,
    padding,
    data_format='NDHWC',
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Args:

- **`input`**: A 5-D `Tensor` of the format specified by `data_format`.
- **`ksize`**: An int or list of `ints` that has length `1`, `3` or `5`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `3` or `5`. The stride of the sliding window for each dimension of the input tensor.

- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: An optional string from: "NDHWC", "NCDHW". Defaults to "NDHWC". The data format of the input and output data. With the default format "NDHWC", the data is stored in the order of: [batch, in_depth, in_height, in_width, in_channels]. Alternatively, the format could be "NCDHW", the data storage order is: [batch, in_channels, in_depth, in_height, in_width].
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of format specified by `data_format`. The max pooled output tensor.

tf.nn.max_pool_with_argmax

- Contents
- Aliases:

Performs max pooling on the input and outputs both max values and indices.

Aliases:

- `tf.compat.v2.nn.max_pool_with_argmax`
- `tf.nn.max_pool_with_argmax`

```
tf.nn.max_pool_with_argmax(
    input,
    ksize,
    strides,
    padding,
    data_format='NHWC',
    output_dtype=tf.dtypes.int64,
    include_batch_in_index=False,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
The indices in `argmax` are flattened, so that a maximum value at position `[b, y, x, c]` becomes flattened index: `(y * width + x) * channels + c` if `include_batch_in_index` is False; `((b * height + y) * width + x) * channels + c` if `include_batch_in_index` is True.
The indices returned are always in `[0, height) x [0, width)` before flattening, even if padding is involved and the mathematically correct answer is outside (either negative or too large). This is a bug, but fixing it is difficult to do in a safe backwards compatible way, especially due to flattening.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`. 4-D with shape `[batch, height, width, channels]`. Input to pool over.
- **`ksize`**: An int or list of `ints` that has length `1`, `2` or `4`. The size of the window for each dimension of the input tensor.
- **`strides`**: An int or list of `ints` that has length `1`, `2` or `4`. The stride of the sliding window for each dimension of the input tensor.
- **`padding`**: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- **`data_format`**: An optional `string`, must be set to `"NHWC"`. Defaults to `"NHWC"`. Specify the data format of the input and output data.
- **`output_dtype`**: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`. The dtype of the returned argmax tensor.

- **`include_batch_in_index`**: An optional `boolean`. Defaults to `False`. Whether to include batch dimension in flattened index of `argmax`.
- **`name`**: A name for the operation (optional).

Returns:
A tuple of `Tensor` objects (output, argmax).
- **`output`**: A `Tensor`. Has the same type as `input`.
- **`argmax`**: A `Tensor` of type `output_dtype`.

tf.nn.moments

- Contents
- Aliases:

Calculates the mean and variance of `x`.

Aliases:
- `tf.compat.v2.nn.moments`
- `tf.nn.moments`

```
tf.nn.moments(
    x,
    axes,
    shift=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.
The mean and variance are calculated by aggregating the contents of `x` across `axes`. If `x` is 1-D and `axes = [0]` this is just the mean and variance of a vector.
Note: shift is currently not used; the true mean is computed and used.
When using these moments for batch normalization (see `tf.nn.batch_normalization`):
- for so-called "global normalization", used with convolutional filters with shape `[batch, height, width, depth]`, pass `axes=[0, 1, 2]`.
- for simple batch normalization pass `axes=[0]` (batch only).

Args:
- **`x`**: A `Tensor`.
- **`axes`**: Array of ints. Axes along which to compute mean and variance.
- **`shift`**: Not used in the current implementation.
- **`keepdims`**: produce moments with the same dimensionality as the input.
- **`name`**: Name used to scope the operations that compute the moments.

Returns:
Two `Tensor` objects: `mean` and `variance`.

tf.nn.nce_loss

- Contents
- Aliases:

Computes and returns the noise-contrastive estimation training loss.

Aliases:
- `tf.compat.v2.nn.nce_loss`
- `tf.nn.nce_loss`

```
tf.nn.nce_loss(
    weights,
    biases,
    labels,
    inputs,
    num_sampled,
    num_classes,
    num_true=1,
    sampled_values=None,
    remove_accidental_hits=False,
    name='nce_loss'
)
```

Defined in `python/ops/nn_impl.py`.
See Noise-contrastive estimation: A new estimation principle for unnormalized statistical models.
Also see our Candidate Sampling Algorithms Reference
A common use case is to use this method for training, and calculate the full sigmoid loss for evaluation or inference as in the following example:

```
if mode == "train":
  loss = tf.nn.nce_loss(
      weights=weights,
      biases=biases,
      labels=labels,
      inputs=inputs,
      ...)
elif mode == "eval":
  logits = tf.matmul(inputs, tf.transpose(weights))
  logits = tf.nn.bias_add(logits, biases)
  labels_one_hot = tf.one_hot(labels, n_classes)
  loss = tf.nn.sigmoid_cross_entropy_with_logits(
      labels=labels_one_hot,
      logits=logits)
  loss = tf.reduce_sum(loss, axis=1)
```

Note: when doing embedding lookup on `weights` and `bias`, "div" partition strategy will be used. Support for other partition strategy will be added later.**Note:** By default this uses a log-uniform (Zipfian) distribution for sampling, so your labels must be sorted in order of decreasing frequency to achieve good results. For more details, see`tf.random.log_uniform_candidate_sampler`.**Note:** In the case where `num_true` > 1, we assign to each target class the target probability 1 / `num_true` so that the target probabilities sum to 1 per-example.**Note:** It would be useful to allow a variable number of target classes per example. We hope to provide this functionality in a future release. For now, if you have a variable number of target classes, you can pad them out to a constant number by either repeating them or by padding with an otherwise unused class.

Args:

- **`weights`**: A `Tensor` of shape `[num_classes, dim]`, or a list of `Tensor` objects whose concatenation along dimension 0 has shape [num_classes, dim]. The (possibly-partitioned) class embeddings.
- **`biases`**: A `Tensor` of shape `[num_classes]`. The class biases.
- **`labels`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.

- **inputs**: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- **num_sampled**: An `int`. The number of negative classes to randomly sample per batch. This single sample of negative classes is evaluated for each element in the batch.
- **num_classes**: An `int`. The number of possible classes.
- **num_true**: An `int`. The number of target classes per training example.
- **sampled_values**: a tuple of (`sampled_candidates`, `true_expected_count`,`sampled_expected_count`) returned by a `*_candidate_sampler` function. (if None, we default to `log_uniform_candidate_sampler`)
- **remove_accidental_hits**: A `bool`. Whether to remove "accidental hits" where a sampled class equals one of the target classes. If set to `True`, this is a "Sampled Logistic" loss instead of NCE, and we are learning to generate log-odds instead of log probabilities. See our Candidate Sampling Algorithms Reference. Default is False.
- **name**: A name for the operation (optional).

Returns:
A `batch_size` 1-D tensor of per-example NCE losses.

tf.nn.normalize_moments

- Contents
- Aliases:

Calculate the mean and variance of based on the sufficient statistics.

Aliases:

- `tf.compat.v1.nn.normalize_moments`
- `tf.compat.v2.nn.normalize_moments`
- `tf.nn.normalize_moments`

```
tf.nn.normalize_moments(
    counts,
    mean_ss,
    variance_ss,
    shift,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

Args:

- **counts**: A `Tensor` containing the total count of the data (one value).
- **mean_ss**: A `Tensor` containing the mean sufficient statistics: the (possibly shifted) sum of the elements to average over.
- **variance_ss**: A `Tensor` containing the variance sufficient statistics: the (possibly shifted) squared sum of the data to compute the variance over.
- **shift**: A `Tensor` containing the value by which the data is shifted for numerical stability, or `None` if no shift was performed.
- **name**: Name used to scope the operations that compute the moments.

Returns:
Two `Tensor` objects: `mean` and `variance`.

tf.nn.pool

- Contents
- Aliases:

Performs an N-D pooling operation.

Aliases:

- `tf.compat.v2.nn.pool`
- `tf.nn.pool`

```
tf.nn.pool(
    input,
    window_shape,
    pooling_type,
    strides=None,
    padding='VALID',
    data_format=None,
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
In the case that `data_format` does not start with "NC", computes for 0 <= b < batch_size, 0 <= x[i] < output_spatial_shape[i], 0 <= c < num_channels:

```
  output[b, x[0], ..., x[N-1], c] =
    REDUCE_{z[0], ..., z[N-1]}
      input[b,
            x[0] * strides[0] - pad_before[0] + dilation_rate[0]*z[0],
            ...
            x[N-1]*strides[N-1] - pad_before[N-1] + dilation_rate[N-1]*z[N-1],
            c],
```

where the reduction function REDUCE depends on the value of `pooling_type`, and pad_before is defined based on the value of `padding` as described in the "returns" section of `tf.nn.convolution`for details. The reduction never includes out-of-bounds positions.
In the case that `data_format` starts with `"NC"`, the `input` and output are simply transposed as follows:

```
  pool(input, data_format, **kwargs) =
    tf.transpose(pool(tf.transpose(input, [0] + range(2,N+2) + [1]),
                      **kwargs),
                 [0, N+1] + range(1, N+1))
```

Args:

- **`input`**: Tensor of rank N+2, of shape `[batch_size] + input_spatial_shape + [num_channels]` if data_format does not start with "NC" (default), or `[batch_size, num_channels] + input_spatial_shape` if data_format starts with "NC". Pooling happens over the spatial dimensions only.
- **`window_shape`**: Sequence of N ints >= 1.
- **`pooling_type`**: Specifies pooling operation, must be "AVG" or "MAX".
- **`strides`**: Optional. Sequence of N ints >= 1. Defaults to [1]*N. If any value of strides is > 1, then all values of dilation_rate must be 1.
- **`padding`**: The padding algorithm, must be "SAME" or "VALID". Defaults to "SAME". See the "returns" section of `tf.nn.convolution` for details.

- **`data_format`**: A string or None. Specifies whether the channel dimension of the `input` and output is the last dimension (default, or if `data_format` does not start with "NC"), or the second dimension (if `data_format` starts with "NC"). For N=1, the valid values are "NWC" (default) and "NCW". For N=2, the valid values are "NHWC" (default) and "NCHW". For N=3, the valid values are "NDHWC" (default) and "NCDHW".
- **`dilations`**: Optional. Dilation rate. List of N ints >= 1. Defaults to [1]*N. If any value of dilation_rate is > 1, then all values of strides must be 1.
- **`name`**: Optional. Name of the op.

Returns:
Tensor of rank N+2, of shape [batch_size] + output_spatial_shape + [num_channels]
if data_format is None or does not start with "NC", or
[batch_size, num_channels] + output_spatial_shape
if data_format starts with "NC", where `output_spatial_shape` depends on the value of padding:
If padding = "SAME": output_spatial_shape[i] = ceil(input_spatial_shape[i] / strides[i])
If padding = "VALID": output_spatial_shape[i] = ceil((input_spatial_shape[i] - (window_shape[i] - 1) * dilation_rate[i]) / strides[i]).

Raises:

- **`ValueError`**: if arguments are invalid.

tf.nn.relu

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes rectified linear: `max(features, 0)`.

Aliases:

- `tf.compat.v1.nn.relu`
- `tf.compat.v2.nn.relu`
- `tf.nn.relu`

```
tf.nn.relu(
    features,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Used in the guide:

- Writing layers and models with TensorFlow Keras
- tf.function and AutoGraph in TensorFlow 2.0

Used in the tutorials:

- Custom layers
- Image Captioning with Attention

Args:

- **`features`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`, `qint8`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has the same type as `features`.

tf.nn.relu6

- Contents
- Aliases:

Computes Rectified Linear 6: `min(max(features, 0), 6)`.

Aliases:

- `tf.compat.v1.nn.relu6`
- `tf.compat.v2.nn.relu6`
- `tf.nn.relu6`

```
tf.nn.relu6(
    features,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
Source: Convolutional Deep Belief Networks on CIFAR-10. A. Krizhevsky

Args:

- **`features`**: A `Tensor` with type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` with the same type as `features`.

tf.nn.RNNCellDeviceWrapper

- Contents
- Class RNNCellDeviceWrapper
  - Aliases:
- __init__
- Properties

Class `RNNCellDeviceWrapper`

Operator that ensures an RNNCell runs on a particular device.

Aliases:

- Class `tf.compat.v2.nn.RNNCellDeviceWrapper`
- Class `tf.nn.RNNCellDeviceWrapper`

Defined in `python/keras/layers/rnn_cell_wrapper_v2.py`.

`__init__`

```
__init__(
    *args,
    **kwargs
)
```

Construct a `DeviceWrapper` for `cell` with device `device`.
Ensures the wrapped `cell` is called with `tf.device(device)`.

Args:

- **`cell`**: An instance of `RNNCell`.

- `device`: A device string or function, for passing to `tf.device`.
- `**kwargs`: dict of keyword arguments for base layer.

Properties

`output_size`

`state_size`

Methods

`get_initial_state`

```
get_initial_state(
    inputs=None,
    batch_size=None,
    dtype=None
)
```

`zero_state`

```
zero_state(
    batch_size,
    dtype
)
```

tf.nn.RNNCellDropoutWrapper

- Contents
- Class RNNCellDropoutWrapper
  - Aliases:
- __init__
- Properties

Class `RNNCellDropoutWrapper`

Operator adding dropout to inputs and outputs of the given cell.

Aliases:

- Class `tf.compat.v2.nn.RNNCellDropoutWrapper`
- Class `tf.nn.RNNCellDropoutWrapper`

Defined in `python/keras/layers/rnn_cell_wrapper_v2.py`.

`__init__`

```
__init__(
    *args,
    **kwargs
)
```

Create a cell with added input, state, and/or output dropout.
If `variational_recurrent` is set to `True` (**NOT** the default behavior), then the same dropout mask is applied at every step, as described in: [A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. Y. Gal, Z. Ghahramani](#).
Otherwise a different dropout mask is applied at every time step.

Note, by default (unless a custom `dropout_state_filter` is provided), the memory state (`c`component of any `LSTMStateTuple`) passing through a `DropoutWrapper` is never modified. This behavior is described in the above article.

Args:

- **`cell`**: an RNNCell, a projection to output_size is added to it.
- **`input_keep_prob`**: unit Tensor or float between 0 and 1, input keep probability; if it is constant and 1, no input dropout will be added.
- **`output_keep_prob`**: unit Tensor or float between 0 and 1, output keep probability; if it is constant and 1, no output dropout will be added.
- **`state_keep_prob`**: unit Tensor or float between 0 and 1, output keep probability; if it is constant and 1, no output dropout will be added. State dropout is performed on the outgoing states of the cell. **Note** the state components to which dropout is applied when `state_keep_prob` is in `(0, 1)` are also determined by the argument `dropout_state_filter_visitor` (e.g. by default dropout is never applied to the `c` component of an `LSTMStateTuple`).
- **`variational_recurrent`**: Python bool. If `True`, then the same dropout pattern is applied across all time steps per run call. If this parameter is set, `input_size` **must** be provided.
- **`input_size`**: (optional) (possibly nested tuple of) `TensorShape` objects containing the depth(s) of the input tensors expected to be passed in to the `DropoutWrapper`. Required and used **iff**`variational_recurrent = True` and `input_keep_prob < 1`.
- **`dtype`**: (optional) The `dtype` of the input, state, and output tensors. Required and used **iff**`variational_recurrent = True`.
- **`seed`**: (optional) integer, the randomness seed.
- **`dropout_state_filter_visitor`**: (optional), default: (see below). Function that takes any hierarchical level of the state and returns a scalar or depth=1 structure of Python booleans describing which terms in the state should be dropped out. In addition, if the function returns `True`, dropout is applied across this sublevel. If the function returns `False`, dropout is not applied across this entire sublevel. Default behavior: perform dropout on all terms except the memory (`c`) state of `LSTMCellState` objects, and don't try to apply dropout to `TensorArray`objects: `def dropout_state_filter_visitor(s): if isinstance(s, LSTMCellState): # Never perform dropout on the c state. return LSTMCellState(c=False, h=True) elif isinstance(s, TensorArray): return False return True`
- **`**kwargs`**: dict of keyword arguments for base layer.

Raises:

- **`TypeError`**: if `cell` is not an `RNNCell`, or `keep_state_fn` is provided but not `callable`.
- **`ValueError`**: if any of the keep_probs are not between 0 and 1.

Properties

`output_size`

`state_size`

`wrapped_cell`

Methods

`get_initial_state`

```
get_initial_state(
    inputs=None,
    batch_size=None,
    dtype=None
```

```
)
```

zero_state

```
zero_state(
    batch_size,
    dtype
)
```

tf.nn.RNNCellResidualWrapper

- Contents
- Class RNNCellResidualWrapper
  - Aliases:
- __init__
- Properties

Class RNNCellResidualWrapper

RNNCell wrapper that ensures cell inputs are added to the outputs.

Aliases:

- Class `tf.compat.v2.nn.RNNCellResidualWrapper`
- Class `tf.nn.RNNCellResidualWrapper`

Defined in `python/keras/layers/rnn_cell_wrapper_v2.py`.

__init__

```
__init__(
    *args,
    **kwargs
)
```

Constructs a `ResidualWrapper` for `cell`.

Args:

- **`cell`**: An instance of `RNNCell`.
- **`residual_fn`**: (Optional) The function to map raw cell inputs and raw cell outputs to the actual cell outputs of the residual network. Defaults to calling nest.map_structure on (lambda i, o: i + o), inputs and outputs.
- **`**kwargs`**: dict of keyword arguments for base layer.

Properties

output_size

state_size

Methods

get_initial_state

```
get_initial_state(
    inputs=None,
    batch_size=None,
    dtype=None
```

```
)
```

zero_state

```
zero_state(
    batch_size,
    dtype
)
```

tf.nn.safe_embedding_lookup_sparse

- Contents
- Aliases:

Lookup embedding results, accounting for invalid IDs and empty features.

Aliases:

- `tf.compat.v2.nn.safe_embedding_lookup_sparse`
- `tf.nn.safe_embedding_lookup_sparse`

```
tf.nn.safe_embedding_lookup_sparse(
    embedding_weights,
    sparse_ids,
    sparse_weights=None,
    combiner='mean',
    default_id=None,
    max_norm=None,
    name=None
)
```

Defined in `python/ops/embedding_ops.py`.

The partitioned embedding in `embedding_weights` must all be the same shape except for the first dimension. The first dimension is allowed to vary as the vocabulary size is not necessarily a multiple of `P`. `embedding_weights` may be a `PartitionedVariable` as returned by using`tf.compat.v1.get_variable()` with a partitioner.

Invalid IDs (< 0) are pruned from input IDs and weights, as well as any IDs with non-positive weight. For an entry with no features, the embedding vector for `default_id` is returned, or the 0-vector if `default_id` is not supplied.

The ids and weights may be multi-dimensional. Embeddings are always aggregated along the last dimension.

Note: when doing embedding lookup on **`embedding_weights`**, "div" partition strategy will be used. Support for other partition strategy will be added later.

Args:

- **`embedding_weights`**: A list of `P` float `Tensor`s or values representing partitioned embedding `Tensor`s. Alternatively, a `PartitionedVariable` created by partitioning along dimension 0. The total unpartitioned shape should be `[e_0, e_1, ..., e_m]`, where `e_0` represents the vocab size and `e_1, ..., e_m` are the embedding dimensions.
- **`sparse_ids`**: `SparseTensor` of shape `[d_0, d_1, ..., d_n]` containing the ids. `d_0` is typically batch size.
- **`sparse_weights`**: `SparseTensor` of same shape as `sparse_ids`, containing float weights corresponding to `sparse_ids`, or `None` if all weights are be assumed to be 1.0.
- **`combiner`**: A string specifying how to combine embedding results for each entry. Currently "mean", "sqrtn" and "sum" are supported, with "mean" the default.

- `default_id`: The id to use for an entry with no features.
- `max_norm`: If not `None`, all embeddings are l2-normalized to max_norm before combining.
- `name`: A name for this operation (optional).

Returns:
Dense `Tensor` of shape `[d_0, d_1, ..., d_{n-1}, e_1, ..., e_m]`.

Raises:

- `ValueError`: if `embedding_weights` is empty.

tf.nn.sampled_softmax_loss

- Contents
- Aliases:

Computes and returns the sampled softmax training loss.

Aliases:

- `tf.compat.v2.nn.sampled_softmax_loss`
- `tf.nn.sampled_softmax_loss`

```
tf.nn.sampled_softmax_loss(
    weights,
    biases,
    labels,
    inputs,
    num_sampled,
    num_classes,
    num_true=1,
    sampled_values=None,
    remove_accidental_hits=True,
    seed=None,
    name='sampled_softmax_loss'
)
```

Defined in `python/ops/nn_impl.py`.
This is a faster way to train a softmax classifier over a huge number of classes.
This operation is for training only. It is generally an underestimate of the full softmax loss.
A common use case is to use this method for training, and calculate the full sigmoid loss for evaluation or inference as in the following example:

```
if mode == "train":
  loss = tf.nn.sampled_softmax_loss(
      weights=weights,
      biases=biases,
      labels=labels,
      inputs=inputs,
      ...)
elif mode == "eval":
  logits = tf.matmul(inputs, tf.transpose(weights))
  logits = tf.nn.bias_add(logits, biases)
  labels_one_hot = tf.one_hot(labels, n_classes)
  loss = tf.nn.softmax_cross_entropy_with_logits(
      labels=labels_one_hot,
```

```
      logits=logits)
```

See our Candidate Sampling Algorithms Reference
Also see Section 3 of Jean et al., 2014 (pdf) for the math.
Note: when doing embedding lookup on **`weights`** and **`bias`**, "div" partition strategy will be used. Support for other partition strategy will be added later.

Args:

- **`weights`**: A `Tensor` of shape `[num_classes, dim]`, or a list of `Tensor` objects whose concatenation along dimension 0 has shape [num_classes, dim]. The (possibly-sharded) class embeddings.
- **`biases`**: A `Tensor` of shape `[num_classes]`. The class biases.
- **`labels`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes. Note that this format differs from the `labels` argument of `nn.softmax_cross_entropy_with_logits`.
- **`inputs`**: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- **`num_sampled`**: An `int`. The number of classes to randomly sample per batch.
- **`num_classes`**: An `int`. The number of possible classes.
- **`num_true`**: An `int`. The number of target classes per training example.
- **`sampled_values`**: a tuple of (`sampled_candidates`, `true_expected_count`,`sampled_expected_count`) returned by a `*_candidate_sampler` function. (if None, we default to `log_uniform_candidate_sampler`)
- **`remove_accidental_hits`**: A `bool`. whether to remove "accidental hits" where a sampled class equals one of the target classes. Default is True.
- **`seed`**: random seed for candidate sampling. Default to None, which doesn't set the op-level random seed for candidate sampling.
- **`name`**: A name for the operation (optional).

Returns:
A `batch_size` 1-D tensor of per-example sampled softmax losses.

tf.nn.scale_regularization_loss

- Contents
- Aliases:

Scales the sum of the given regularization losses by number of replicas.

Aliases:

- `tf.compat.v1.nn.scale_regularization_loss`
- `tf.compat.v2.nn.scale_regularization_loss`
- `tf.nn.scale_regularization_loss`

```
tf.nn.scale_regularization_loss(regularization_loss)
```

Defined in `python/ops/nn_impl.py`.
Usage with distribution strategy and custom training loop:

```
with strategy.scope():
  def compute_loss(self, label, predictions):
    per_example_loss = tf.keras.losses.sparse_categorical_crossentropy(
        labels, predictions)

    # Compute loss that is scaled by sample_weight and by global batch size.
    loss = tf.compute_average_loss(
        per_example_loss,
        sample_weight=sample_weight,
```

```
        global_batch_size=GLOBAL_BATCH_SIZE)

    # Add scaled regularization losses.
    loss += tf.scale_regularization_loss(tf.nn.l2_loss(weights))
    return loss
```

Args:

- **`regularization_loss`**: Regularization loss.

Returns:
Scalar loss value.

tf.nn.selu

- Contents
- Aliases:

Computes scaled exponential linear: `scale * alpha * (exp(features) - 1)`

Aliases:

- `tf.compat.v1.nn.selu`
- `tf.compat.v2.nn.selu`
- `tf.nn.selu`

```
tf.nn.selu(
    features,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.
if < 0, `scale * features` otherwise.
To be used together with `initializer = tf.variance_scaling_initializer(factor=1.0, mode='FAN_IN')`. For correct dropout, use `tf.contrib.nn.alpha_dropout`.
See Self-Normalizing Neural Networks

Args:

- **`features`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has the same type as `features`.

tf.nn.separable_conv2d

- Contents
- Aliases:

2-D convolution with separable filters.

Aliases:

- `tf.compat.v2.nn.separable_conv2d`
- `tf.nn.separable_conv2d`

```
tf.nn.separable_conv2d(
    input,
    depthwise_filter,
    pointwise_filter,
    strides,
```

```
    padding,
    data_format=None,
    dilations=None,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.
Performs a depthwise convolution that acts separately on channels followed by a pointwise convolution that mixes channels. Note that this is separability between dimensions `[1, 2]` and `3`, not spatial separability between dimensions `1` and `2`.
In detail, with the default NHWC format,

```
output[b, i, j, k] = sum_{di, dj, q, r}
    input[b, strides[1] * i + di, strides[2] * j + dj, q] *
    depthwise_filter[di, dj, q, r] *
    pointwise_filter[0, 0, q * channel_multiplier + r, k]
```

`strides` controls the strides for the depthwise convolution only, since the pointwise convolution has implicit strides of `[1, 1, 1, 1]`. Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`. If any value in `rate` is greater than 1, we perform atrous depthwise convolution, in which case all values in the `strides` tensor must be equal to 1.

Args:

- **`input`**: 4-D `Tensor` with shape according to `data_format`.
- **`depthwise_filter`**: 4-D `Tensor` with shape `[filter_height, filter_width, in_channels, channel_multiplier]`. Contains `in_channels` convolutional filters of depth 1.
- **`pointwise_filter`**: 4-D `Tensor` with shape `[1, 1, channel_multiplier * in_channels, out_channels]`. Pointwise filter to mix channels after `depthwise_filter` has convolved spatially.
- **`strides`**: 1-D of size 4. The strides for the depthwise convolution for each dimension of `input`.
- **`padding`**: A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the "returns" section of `tf.nn.convolution` for details.
- **`data_format`**: The data format for input. Either "NHWC" (default) or "NCHW".
- **`dilations`**: 1-D of size 2. The dilation rate in which we sample input values across the `height`and `width` dimensions in atrous convolution. If it is greater than 1, then all values of strides must be 1.
- **`name`**: A name for this operation (optional).

Returns:

A 4-D `Tensor` with shape according to 'data_format'. For example, with data_format="NHWC", shape is [batch, out_height, out_width, out_channels].

tf.nn.sigmoid_cross_entropy_with_logits

- Contents
- Aliases:
- Used in the tutorials:

Computes sigmoid cross entropy given `logits`.

Aliases:

- `tf.compat.v2.nn.sigmoid_cross_entropy_with_logits`
- `tf.nn.sigmoid_cross_entropy_with_logits`

```
tf.nn.sigmoid_cross_entropy_with_logits(
    labels=None,
    logits=None,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

Used in the tutorials:

- Convolutional Variational Autoencoder

Measures the probability error in discrete classification tasks in which each class is independent and not mutually exclusive. For instance, one could perform multilabel classification where a picture can contain both an elephant and a dog at the same time.

For brevity, let `x = logits`, `z = labels`. The logistic loss is

```
  z * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))
= z * -log(1 / (1 + exp(-x))) + (1 - z) * -log(exp(-x) / (1 + exp(-x)))
= z * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-x)))
= z * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x))
= (1 - z) * x + log(1 + exp(-x))
= x - x * z + log(1 + exp(-x))
```

For x < 0, to avoid overflow in exp(-x), we reformulate the above

```
  x - x * z + log(1 + exp(-x))
= log(exp(x)) - x * z + log(1 + exp(-x))
= - x * z + log(1 + exp(x))
```

Hence, to ensure stability and avoid overflow, the implementation uses this equivalent formulation

```
max(x, 0) - x * z + log(1 + exp(-abs(x)))
```

`logits` and `labels` must have the same type and shape.

Args:

- **`labels`**: A `Tensor` of the same type and shape as `logits`.
- **`logits`**: A `Tensor` of type `float32` or `float64`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `logits` with the componentwise logistic losses.

Raises:

- **`ValueError`**: If `logits` and `labels` do not have the same shape.

tf.nn.softmax

- Contents
- Aliases:
- Used in the tutorials:

Computes softmax activations.

Aliases:

- `tf.compat.v2.math.softmax`
- `tf.compat.v2.nn.softmax`
- `tf.math.softmax`

- `tf.nn.softmax`

```
tf.nn.softmax(
    logits,
    axis=None,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Used in the tutorials:

- [Custom training: walkthrough]
- [Image Captioning with Attention]
- [Neural Machine Translation with Attention]
- [Transformer model for language understanding]

This function performs the equivalent of

```
softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis)
```

Args:

- **`logits`**: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- **`axis`**: The dimension softmax would be performed on. The default is -1 which indicates the last dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type and shape as `logits`.

Raises:

- **`InvalidArgumentError`**: if `logits` is empty or `axis` is beyond the last dimension of `logits`.

tf.nn.softmax_cross_entropy_with_logits

- Contents
- Aliases:
- Used in the guide:

Computes softmax cross entropy between `logits` and `labels`.

Aliases:

- `tf.compat.v2.nn.softmax_cross_entropy_with_logits`
- `tf.nn.softmax_cross_entropy_with_logits`

```
tf.nn.softmax_cross_entropy_with_logits(
    labels,
    logits,
    axis=-1,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Used in the guide:

- [Distributed training in TensorFlow]

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE: While the classes are mutually exclusive, their probabilities need not be. All that is required is that each row of `labels` is a valid probability distribution. If they are not, the computation of the gradient will be incorrect.
If using exclusive `labels` (wherein one and only one class is true at a time), see `sparse_softmax_cross_entropy_with_logits`.
WARNING: This op expects unscaled logits, since it performs a `softmax` on `logits` internally for efficiency. Do not call this op with the output of `softmax`, as it will produce incorrect results.
A common use case is to have logits and labels of shape `[batch_size, num_classes]`, but higher dimensions are supported, with the `axis` argument specifying the class dimension.
`logits` and `labels` must have the same dtype (either `float16`, `float32`, or `float64`).
Backpropagation will happen into both `logits` and `labels`. To disallow backpropagation into `labels`, pass label tensors through `tf.stop_gradient` before feeding it to this function.
Note that to avoid confusion, it is required to pass only named arguments to this function.

Args:

- **`labels`**: Each vector along the class dimension should hold a valid probability distribution e.g. for the case in which labels are of shape `[batch_size, num_classes]`, each row of `labels[i]` must be a valid probability distribution.
- **`logits`**: Per-label activations, typically a linear output. These activation energies are interpreted as unnormalized log probabilities.
- **`axis`**: The class dimension. Defaulted to -1 which is the last dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` that contains the softmax cross entropy loss. Its type is the same as `logits` and its shape is the same as `labels` except that it does not have the last dimension of `labels`.

tf.nn.softsign

- Contents
- Aliases:

Computes softsign: `features / (abs(features) + 1)`.

Aliases:

- `tf.compat.v1.math.softsign`
- `tf.compat.v1.nn.softsign`
- `tf.compat.v2.math.softsign`
- `tf.compat.v2.nn.softsign`
- `tf.math.softsign`
- `tf.nn.softsign`

```
tf.nn.softsign(
    features,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Args:

- **`features`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.space_to_depth

- Contents
- Aliases:

SpaceToDepth for tensors of type T.

Aliases:

- `tf.compat.v2.nn.space_to_depth`
- `tf.nn.space_to_depth`

```
tf.nn.space_to_depth(
    input,
    block_size,
    data_format='NHWC',
    name=None
)
```

Defined in `python/ops/array_ops.py`.

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.

- Non-overlapping blocks of size `block_size x block size` are rearranged into depth at each location.
- The depth of the output tensor is `block_size * block_size * input_depth`.
- The Y, X coordinates within each block of the input become the high order component of the output channel index.
- The input tensor's height and width must be divisible by block_size.

The `data_format` attr specifies the layout of the input and output tensors with the following options: "NHWC": `[ batch, height, width, channels ]` "NCHW": `[ batch, channels, height, width ]` "NCHW_VECT_C": `qint8 [ batch, channels / 4, height, width, 4 ]`

It is useful to consider the operation as transforming a 6-D Tensor. e.g. for data_format = NHWC, Each element in the input tensor can be specified via 6 coordinates, ordered by decreasing memory layout significance as: n,oY,bY,oX,bX,iC (where n=batch index, oX, oY means X or Y coordinates within the output image, bX, bY means coordinates within the input block, iC means input channels). The output would be a transpose to the following layout: n,oY,oX,bY,bX,iC

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 an input of shape `[1, 2, 2, 1]`, data_format = "NHWC" and block_size = 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.
- **`data_format`**: An optional `string` from: `"NHWC", "NCHW", "NCHW_VECT_C"`. Defaults to `"NHWC"`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.nn.sparse_softmax_cross_entropy_with_logits

- Contents
- Aliases:

Computes sparse softmax cross entropy between `logits` and `labels`.

Aliases:

- `tf.compat.v2.nn.sparse_softmax_cross_entropy_with_logits`
- `tf.nn.sparse_softmax_cross_entropy_with_logits`

```
tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels,
    logits,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE: For this operation, the probability of a given label is considered exclusive. That is, soft classes are not allowed, and the `labels` vector must provide a single specific index for the true class for each row of `logits` (each minibatch entry). For soft softmax classification with a probability distribution for each entry, see `softmax_cross_entropy_with_logits_v2`.

WARNING: This op expects unscaled logits, since it performs a `softmax` on `logits` internally for efficiency. Do not call this op with the output of `softmax`, as it will produce incorrect results.

A common use case is to have logits of shape `[batch_size, num_classes]` and have labels of shape `[batch_size]`, but higher dimensions are supported, in which case the `dim`-th dimension is

assumed to be of size `num_classes`. `logits` must have the dtype of `float16`, `float32`, or `float64`, and `labels` must have the dtype of `int32` or `int64`.
Note that to avoid confusion, it is required to pass only named arguments to this function.

Args:

- **`labels`**: `Tensor` of shape `[d_0, d_1, ..., d_{r-1}]` (where `r` is rank of `labels` and result) and dtype `int32` or `int64`. Each entry in `labels` must be an index in `[0, num_classes)`. Other values will raise an exception when this op is run on CPU, and return `NaN`for corresponding loss and gradient rows on GPU.
- **`logits`**: Unscaled log probabilities of shape `[d_0, d_1, ..., d_{r-1}, num_classes]` and dtype `float16`, `float32`, or `float64`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `labels` and of the same type as `logits` with the softmax cross entropy loss.

Raises:

- **`ValueError`**: If logits are scalars (need to have rank >= 1) or if the rank of the labels is not equal to the rank of the logits minus one.

tf.nn.sufficient_statistics

- Contents
- Aliases:

Calculate the sufficient statistics for the mean and variance of `x`.

Aliases:

- `tf.compat.v2.nn.sufficient_statistics`
- `tf.nn.sufficient_statistics`

```
tf.nn.sufficient_statistics(
    x,
    axes,
    shift=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.
These sufficient statistics are computed using the one pass algorithm on an input that's optionally shifted. See:
https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Computing_shifted_data

Args:

- **`x`**: A `Tensor`.
- **`axes`**: Array of ints. Axes along which to compute mean and variance.
- **`shift`**: A `Tensor` containing the value by which to shift the data for numerical stability, or `None`if no shift is to be performed. A shift close to the true mean provides the most numerically stable results.
- **`keepdims`**: produce statistics with the same dimensionality as the input.
- **`name`**: Name used to scope the operations that compute the sufficient stats.

Returns:

Four `Tensor` objects of the same type as `x`:

- the count (number of elements to average over).
- the (possibly shifted) sum of the elements in the array.

- the (possibly shifted) sum of squares of the elements in the array.
- the shift by which the mean must be corrected or None if `shift` is None.

tf.nn.weighted_cross_entropy_with_logits

- Contents
- Aliases:

Computes a weighted cross entropy.

Aliases:

- `tf.compat.v2.nn.weighted_cross_entropy_with_logits`
- `tf.nn.weighted_cross_entropy_with_logits`

```
tf.nn.weighted_cross_entropy_with_logits(
    labels,
    logits,
    pos_weight,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

This is like `sigmoid_cross_entropy_with_logits()` except that `pos_weight`, allows one to trade off recall and precision by up- or down-weighting the cost of a positive error relative to a negative error.

The usual cross-entropy cost is defined as:

```
labels * -log(sigmoid(logits)) +
    (1 - labels) * -log(1 - sigmoid(logits))
```

A value `pos_weights > 1` decreases the false negative count, hence increasing the recall. Conversely setting `pos_weights < 1` decreases the false positive count and increases the precision. This can be seen from the fact that `pos_weight` is introduced as a multiplicative coefficient for the positive labels term in the loss expression:

```
labels * -log(sigmoid(logits)) * pos_weight +
    (1 - labels) * -log(1 - sigmoid(logits))
```

For brevity, let `x = logits`, `z = labels`, `q = pos_weight`. The loss is:

```
  qz * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))
= qz * -log(1 / (1 + exp(-x))) + (1 - z) * -log(exp(-x) / (1 + exp(-x)))
= qz * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-x)))
= qz * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x))
= (1 - z) * x + (qz +  1 - z) * log(1 + exp(-x))
= (1 - z) * x + (1 + (q - 1) * z) * log(1 + exp(-x))
```

Setting `l = (1 + (q - 1) * z)`, to ensure stability and avoid overflow, the implementation uses

```
(1 - z) * x + l * (log(1 + exp(-abs(x))) + max(-x, 0))
```

`logits` and `labels` must have the same type and shape.

Args:

- **`labels`**: A `Tensor` of the same type and shape as `logits`.
- **`logits`**: A `Tensor` of type `float32` or `float64`.
- **`pos_weight`**: A coefficient to use on the positive examples.

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

  Returns:
  A `Tensor` of the same shape as `logits` with the componentwise weighted logistic losses.

  Raises:
- **`ValueError`**: If `logits` and `labels` do not have the same shape.

tf.nn.weighted_moments

- Contents
- Aliases:

Returns the frequency-weighted mean and variance of `x`.

Aliases:

- `tf.compat.v2.nn.weighted_moments`
- `tf.nn.weighted_moments`

```
tf.nn.weighted_moments(
    x,
    axes,
    frequency_weights,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

Args:

- **`x`**: A tensor.
- **`axes`**: 1-d tensor of int32 values; these are the axes along which to compute mean and variance.
- **`frequency_weights`**: A tensor of positive weights which can be broadcast with x.
- **`keepdims`**: Produce moments with the same dimensionality as the input.
- **`name`**: Name used to scope the operation.

Returns:
Two tensors: `weighted_mean` and `weighted_variance`.

tf.nn.with_space_to_batch

- Contents
- Aliases:

Performs `op` on the space-to-batch representation of `input`.

Aliases:

- `tf.compat.v1.nn.with_space_to_batch`
- `tf.compat.v2.nn.with_space_to_batch`
- `tf.nn.with_space_to_batch`

```
tf.nn.with_space_to_batch(
    input,
    dilation_rate,
    padding,
    op,
    filter_shape=None,
    spatial_dims=None,
    data_format=None
```

```
)
```

Defined in `python/ops/nn_ops.py`.
This has the effect of transforming sliding window operations into the corresponding "atrous" operation in which the input is sampled at the specified `dilation_rate`.
In the special case that `dilation_rate` is uniformly 1, this simply returns:
op(input, num_spatial_dims, padding)
Otherwise, it returns:
batch_to_space_nd(op(space_to_batch_nd(input, adjusted_dilation_rate, adjusted_paddings), num_spatial_dims, "VALID") adjusted_dilation_rate, adjusted_crops),
where:
adjusted_dilation_rate is an int64 tensor of shape [max(spatial*dims)], adjusted*{paddings,crops} are int64 tensors of shape [max(spatial_dims), 2]
defined as follows:
We first define two int64 tensors `paddings` and `crops` of shape `[num_spatial_dims, 2]` based on the value of `padding` and the spatial dimensions of the `input`:
If `padding = "VALID"`, then:
paddings, crops = required_space_to_batch_paddings(input_shape[spatial_dims], dilation_rate)
If `padding = "SAME"`, then:
dilated_filter_shape = filter_shape + (filter_shape - 1) * (dilation_rate - 1)
paddings, crops = required_space_to_batch_paddings(input_shape[spatial_dims], dilation_rate, [(dilated_filter_shape - 1) // 2, dilated_filter_shape - 1 - (dilated_filter_shape - 1) // 2])
Because `space_to_batch_nd` and `batch_to_space_nd` assume that the spatial dimensions are contiguous starting at the second dimension, but the specified `spatial_dims` may not be, we must adjust `dilation_rate`, `paddings` and `crops` in order to be usable with these operations. For a given dimension, if the block size is 1, and both the starting and ending padding and crop amounts are 0, then space_to_batch_nd effectively leaves that dimension alone, which is what is needed for dimensions not part of `spatial_dims`.
Furthermore, `space_to_batch_nd` and `batch_to_space_nd`handle this case efficiently for any number of leading and trailing dimensions.
For 0 <= i < len(spatial_dims), we assign:
adjusted_dilation_rate[spatial_dims[i] - 1] = dilation_rate[i] adjusted_paddings[spatial_dims[i] - 1, :] = paddings[i, :] adjusted_crops[spatial_dims[i] - 1, :] = crops[i, :]
All unassigned values of `adjusted_dilation_rate` default to 1, while all unassigned values of `adjusted_paddings` and `adjusted_crops` default to 0.
Note in the case that `dilation_rate` is not uniformly 1, specifying "VALID" padding is equivalent to specifying `padding = "SAME"` with a filter_shape of `[1]*N`.
Advanced usage. Note the following optimization: A sequence of `with_space_to_batch` operations with identical (not uniformly 1) `dilation_rate` parameters and "VALID" padding
net = with_space_to_batch(net, dilation_rate, "VALID", op_1) ... net = with_space_to_batch(net, dilation_rate, "VALID", op_k)
can be combined into a single `with_space_to_batch` operation as follows:
def combined_op(converted_input, num_spatial_dims, _): result = op_1(converted_input, num_spatial_dims, "VALID") ... result = op_k(result, num_spatial_dims, "VALID")
net = with_space_to_batch(net, dilation_rate, "VALID", combined_op)
This eliminates the overhead of `k-1` calls to `space_to_batch_nd` and `batch_to_space_nd`.
Similarly, a sequence of `with_space_to_batch` operations with identical (not uniformly 1) `dilation_rate` parameters, "SAME" padding, and odd filter dimensions
net = with_space_to_batch(net, dilation_rate, "SAME", op_1, filter_shape_1) ... net = with_space_to_batch(net, dilation_rate, "SAME", op_k, filter_shape_k)
can be combined into a single `with_space_to_batch` operation as follows:

def combined_op(converted_input, num_spatial_dims, _): result = op_1(converted_input, num_spatial_dims, "SAME") ... result = op_k(result, num_spatial_dims, "SAME")
net = with_space_to_batch(net, dilation_rate, "VALID", combined_op)

Args:

- **`input`**: Tensor of rank > max(spatial_dims).
- **`dilation_rate`**: int32 Tensor of *known* shape [num_spatial_dims].
- **`padding`**: str constant equal to "VALID" or "SAME"
- **`op`**: Function that maps (input, num_spatial_dims, padding) -> output
- **`filter_shape`**: If padding = "SAME", specifies the shape of the convolution kernel/pooling window as an integer Tensor of shape [>=num_spatial_dims]. If padding = "VALID", filter_shape is ignored and need not be specified.
- **`spatial_dims`**: Monotonically increasing sequence of `num_spatial_dims` integers (which are >= 1) specifying the spatial dimensions of `input` and output. Defaults to: `range(1, num_spatial_dims+1)`.
- **`data_format`**: A string or None. Specifies whether the channel dimension of the `input` and output is the last dimension (default, or if `data_format` does not start with "NC"), or the second dimension (if `data_format` starts with "NC"). For N=1, the valid values are "NWC" (default) and "NCW". For N=2, the valid values are "NHWC" (default) and "NCHW". For N=3, the valid values are "NDHWC" (default) and "NCDHW".

Returns:

The output Tensor as described above, dimensions will vary based on the op provided.

Raises:

- **`ValueError`**: if `padding` is invalid or the arguments are incompatible.
- **`ValueError`**: if `spatial_dims` are invalid.

Module: tf.compat.v1.ragged / tf.ragged

- **Contents**
- Additional ops that support RaggedTensor
- Classes
- Functions

Ragged Tensors.

This package defines ops for manipulating ragged tensors (`tf.RaggedTensor`), which are tensors with non-uniform shapes. In particular, each `RaggedTensor` has one or more *ragged dimensions*, which are dimensions whose slices may have different lengths. For example, the inner (column) dimension of `rt=[[3, 1, 4, 1], [], [5, 9, 2], [6], []]` is ragged, since the column slices (`rt[0, :]`, ..., `rt[4, :]`) have different lengths. For a more detailed description of ragged tensors, see the `tf.RaggedTensor` class documentation and the Ragged Tensor Guide.

Additional ops that support `RaggedTensor`

Arguments that accept `RaggedTensor`s are marked in **bold**.

- `tf.batch_gather`(**params**, **indices**, name=`None`)
- `tf.bitwise.bitwise_and`(**x**, **y**, name=`None`)
- `tf.bitwise.bitwise_or`(**x**, **y**, name=`None`)
- `tf.bitwise.bitwise_xor`(**x**, **y**, name=`None`)
- `tf.bitwise.invert`(**x**, name=`None`)
- `tf.bitwise.left_shift`(**x**, **y**, name=`None`)
- `tf.bitwise.right_shift`(**x**, **y**, name=`None`)
- `tf.clip_by_value`(**t**, clip_value_min, clip_value_max, name=`None`)
- `tf.concat`(**values**, axis, name=`'concat'`)
- `tf.debugging.check_numerics`(**tensor**, message, name=`None`)

- `tf.dtypes.cast`(**x**, dtype, name=`None`)
- `tf.dtypes.complex`(**real**, **imag**, name=`None`)
- `tf.dtypes.saturate_cast`(**value**, dtype, name=`None`)
- `tf.expand_dims`(**input**, axis=`None`, name=`None`, dim=`None`)
- `tf.gather_nd`(**params**, **indices**, name=`None`, batch_dims=`0`)
- `tf.gather`(**params**, **indices**, validate_indices=`None`, name=`None`, axis=`None`, batch_dims=`0`)
- `tf.identity`(**input**, name=`None`)
- `tf.io.decode_base64`(**input**, name=`None`)
- `tf.io.decode_compressed`(**bytes**, compression_type=`''`, name=`None`)
- `tf.io.encode_base64`(**input**, pad=`False`, name=`None`)
- `tf.math.abs`(**x**, name=`None`)
- `tf.math.acos`(**x**, name=`None`)
- `tf.math.acosh`(**x**, name=`None`)
- `tf.math.add_n`(**inputs**, name=`None`)
- `tf.math.add`(**x**, **y**, name=`None`)
- `tf.math.angle`(**input**, name=`None`)
- `tf.math.asin`(**x**, name=`None`)
- `tf.math.asinh`(**x**, name=`None`)
- `tf.math.atan2`(**y**, **x**, name=`None`)
- `tf.math.atan`(**x**, name=`None`)
- `tf.math.atanh`(**x**, name=`None`)
- `tf.math.ceil`(**x**, name=`None`)
- `tf.math.conj`(**x**, name=`None`)
- `tf.math.cos`(**x**, name=`None`)
- `tf.math.cosh`(**x**, name=`None`)
- `tf.math.digamma`(**x**, name=`None`)
- `tf.math.divide_no_nan`(**x**, **y**, name=`None`)
- `tf.math.divide`(**x**, **y**, name=`None`)
- `tf.math.equal`(**x**, **y**, name=`None`)
- `tf.math.erf`(**x**, name=`None`)
- `tf.math.erfc`(**x**, name=`None`)
- `tf.math.exp`(**x**, name=`None`)
- `tf.math.expm1`(**x**, name=`None`)
- `tf.math.floor`(**x**, name=`None`)
- `tf.math.floordiv`(**x**, **y**, name=`None`)
- `tf.math.floormod`(**x**, **y**, name=`None`)
- `tf.math.greater_equal`(**x**, **y**, name=`None`)
- `tf.math.greater`(**x**, **y**, name=`None`)
- `tf.math.imag`(**input**, name=`None`)
- `tf.math.is_finite`(**x**, name=`None`)
- `tf.math.is_inf`(**x**, name=`None`)
- `tf.math.is_nan`(**x**, name=`None`)
- `tf.math.less_equal`(**x**, **y**, name=`None`)
- `tf.math.less`(**x**, **y**, name=`None`)
- `tf.math.lgamma`(**x**, name=`None`)
- `tf.math.log1p`(**x**, name=`None`)
- `tf.math.log_sigmoid`(**x**, name=`None`)
- `tf.math.log`(**x**, name=`None`)
- `tf.math.logical_and`(**x**, **y**, name=`None`)
- `tf.math.logical_not`(**x**, name=`None`)

- tf.math.logical_or(**x**, **y**, name=None)
- tf.math.logical_xor(**x**, **y**, name='LogicalXor')
- tf.math.maximum(**x**, **y**, name=None)
- tf.math.minimum(**x**, **y**, name=None)
- tf.math.multiply(**x**, **y**, name=None)
- tf.math.negative(**x**, name=None)
- tf.math.not_equal(**x**, **y**, name=None)
- tf.math.pow(**x**, **y**, name=None)
- tf.math.real(**input**, name=None)
- tf.math.reciprocal(**x**, name=None)
- tf.math.reduce_any(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.reduce_max(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.reduce_mean(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.reduce_min(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.reduce_prod(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.reduce_sum(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.math.rint(**x**, name=None)
- tf.math.round(**x**, name=None)
- tf.math.rsqrt(**x**, name=None)
- tf.math.sign(**x**, name=None)
- tf.math.sin(**x**, name=None)
- tf.math.sinh(**x**, name=None)
- tf.math.sqrt(**x**, name=None)
- tf.math.square(**x**, name=None)
- tf.math.squared_difference(**x**, **y**, name=None)
- tf.math.subtract(**x**, **y**, name=None)
- tf.math.tan(**x**, name=None)
- tf.math.truediv(**x**, **y**, name=None)
- tf.math.unsorted_segment_max(**data**, **segment_ids**, num_segments, name=None)
- tf.math.unsorted_segment_mean(**data**, **segment_ids**, num_segments, name=None)
- tf.math.unsorted_segment_min(**data**, **segment_ids**, num_segments, name=None)
- tf.math.unsorted_segment_prod(**data**, **segment_ids**, num_segments, name=None)
- tf.math.unsorted_segment_sqrt_n(**data**, **segment_ids**, num_segments, name=None)
- tf.math.unsorted_segment_sum(**data**, **segment_ids**, num_segments, name=None)
- tf.ones_like(**tensor**, dtype=None, name=None, optimize=True)
- tf.rank(**input**, name=None)
- tf.realdiv(**x**, **y**, name=None)
- tf.reduce_all(**input_tensor**, axis=None, keepdims=False, name=None)
- tf.size(**input**, name=None, out_type=tf.int32)
- tf.squeeze(**input**, axis=None, name=None, squeeze_dims=None)
- tf.stack(**values**, axis=0, name='stack')
- tf.strings.as_string(**input**, precision=-1, scientific=False, shortest=False, width=-1, fill='', name=None)
- tf.strings.join(**inputs**, separator='', name=None)
- tf.strings.length(**input**, name=None, unit='BYTE')
- tf.strings.regex_full_match(**input**, pattern, name=None)
- tf.strings.regex_replace(**input**, pattern, rewrite, replace_global=True, name=None)
- tf.strings.strip(**input**, name=None)
- tf.strings.substr(**input**, pos, len, name=None, unit='BYTE')
- tf.strings.to_hash_bucket_fast(**input**, num_buckets, name=None)

- `tf.strings.to_hash_bucket_strong`(**input**, num_buckets, key, name=`None`)
- `tf.strings.unicode_script`(**input**, name=`None`)
- `tf.tile`(**input**, multiples, name=`None`)
- `tf.truncatediv`(**x**, **y**, name=`None`)
- `tf.truncatemod`(**x**, **y**, name=`None`)
- `tf.where`(**condition**, **x**=`None`, **y**=`None`, name=`None`)
- `tf.zeros_like`(**tensor**, dtype=`None`, name=`None`, optimize=`True`)n

Classes

`class RaggedTensorValue`: Represents the value of a `RaggedTensor`.

Functions

`boolean_mask(...)`: Applies a boolean mask to `data` without flattening the mask dimensions.
`constant(...)`: Constructs a constant RaggedTensor from a nested Python list.
`constant_value(...)`: Constructs a RaggedTensorValue from a nested Python list.
`map_flat_values(...)`: Applies `op` to the values of one or more RaggedTensors.
`placeholder(...)`: Creates a placeholder for a `tf.RaggedTensor` that will always be fed.
`range(...)`: Returns a `RaggedTensor` containing the specified sequences of numbers.
`row_splits_to_segment_ids(...)`: Generates the segmentation corresponding to a RaggedTensor `row_splits`.
`segment_ids_to_row_splits(...)`: Generates the RaggedTensor `row_splits` corresponding to a segmentation.

tf.compat.v1.ragged.constant_value

Constructs a RaggedTensorValue from a nested Python list.

```
tf.compat.v1.ragged.constant_value(
    pylist,
    dtype=None,
    ragged_rank=None,
    inner_shape=None,
    row_splits_dtype='int64'
)
```

Defined in `python/ops/ragged/ragged_factory_ops.py`.
Warning: This function returns a **`RaggedTensorValue`**, not a **`RaggedTensor`**. If you wish to construct a constant **`RaggedTensor`**, use **`ragged.constant(...)`** instead.

Example:

```
>>> ragged.constant_value([[1, 2], [3], [4, 5, 6]])
RaggedTensorValue(values=[1, 2, 3, 4, 5, 6], splits=[0, 2, 3, 6])
```

All scalar values in `pylist` must have the same nesting depth `K`, and the returned `RaggedTensorValue` will have rank `K`. If `pylist` contains no scalar values, then `K` is one greater than the maximum depth of empty lists in `pylist`. All scalar values in `pylist` must be compatible with `dtype`.

Args:

- **`pylist`**: A nested `list`, `tuple` or `np.ndarray`. Any nested element that is not a `list` or `tuple` must be a scalar value compatible with `dtype`.
- **`dtype`**: `numpy.dtype`. The type of elements for the returned `RaggedTensor`. If not specified, then a default is chosen based on the scalar values in `pylist`.

- **`ragged_rank`**: An integer specifying the ragged rank of the returned `RaggedTensorValue`. Must be nonnegative and less than `K`. Defaults to `max(0, K - 1)` if `inner_shape` is not specified. Defaults to `max(0, K
- 1 - len(inner_shape))`if`inner_shape` is specified.
- **`inner_shape`**: A tuple of integers specifying the shape for individual inner values in the returned `RaggedTensorValue`. Defaults to `()` if `ragged_rank` is not specified. If `ragged_rank` is specified, then a default is chosen based on the contents of `pylist`.
- **`row_splits_dtype`**: data type for the constructed `RaggedTensorValue`'s row_splits. One of `numpy.int32` or `numpy.int64`.

Returns:

A `tf.RaggedTensorValue` or `numpy.array` with rank `K` and the specified `ragged_rank`, containing the values from `pylist`.

Raises:

- **`ValueError`**: If the scalar values in `pylist` have inconsistent nesting depth; or if ragged_rank or inner_shape are incompatible with `pylist`.

tf.compat.v1.ragged.placeholder

Creates a placeholder for a `tf.RaggedTensor` that will always be fed.

```
tf.compat.v1.ragged.placeholder(
    dtype,
    ragged_rank,
    value_shape=None,
    name=None
)
```

Defined in `python/ops/ragged/ragged_factory_ops.py`.

Important: This ragged tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`. @compatibility{eager} Placeholders are not compatible with eager execution.

Args:

- **`dtype`**: The data type for the `RaggedTensor`.
- **`ragged_rank`**: The ragged rank for the `RaggedTensor`
- **`value_shape`**: The shape for individual flat values in the `RaggedTensor`.
- **`name`**: A name for the operation (optional).

Returns:

A `RaggedTensor` that may be used as a handle for feeding a value, but not evaluated directly.

Raises:

- **`RuntimeError`**: if eager execution is enabled

tf.compat.v1.ragged.RaggedTensorValue

- **Contents**

- Class RaggedTensorValue
- __init__
- Properties
  - dtype

Class `RaggedTensorValue`

Represents the value of a `RaggedTensor`.

Defined in `python/ops/ragged/ragged_tensor_value.py`.

Warning: `RaggedTensorValue` should only be used in graph mode; in eager mode, the `tf.RaggedTensor` class contains its value directly.
See `tf.RaggedTensor` for a description of ragged tensors.

`__init__`

```
__init__(
    values,
    row_splits
)
```

Creates a `RaggedTensorValue`.

Args:

- **`values`**: A numpy array of any type and shape; or a RaggedTensorValue.
- **`row_splits`**: A 1-D int32 or int64 numpy array.

Properties

`dtype`

The numpy dtype of values in this tensor.

`flat_values`

The innermost `values` array for this ragged tensor value.

`nested_row_splits`

The row_splits for all ragged dimensions in this ragged tensor value.

`ragged_rank`

The number of ragged dimensions in this ragged tensor value.

`row_splits`

The split indices for the ragged tensor value.

`shape`

A tuple indicating the shape of this RaggedTensorValue.

`values`

The concatenated values for all rows in this tensor.

Methods

`to_list`

```
to_list()
```

Returns this ragged tensor value as a nested Python list.

tf.ragged.boolean_mask

- **Contents**
- Aliases:

Applies a boolean mask to `data` without flattening the mask dimensions.

Aliases:

- `tf.compat.v1.ragged.boolean_mask`
- `tf.compat.v2.ragged.boolean_mask`
- `tf.ragged.boolean_mask`

```
tf.ragged.boolean_mask(
    data,
    mask,
    name=None
)
```

Defined in python/ops/ragged/ragged_array_ops.py.
Returns a potentially ragged tensor that is formed by retaining the elements in `data` where the corresponding value in `mask` is `True`.

- `output[a1...aA, i, b1...bB] = data[a1...aA, j, b1...bB]`

Where `j` is the `i`th `True` entry of `mask[a1...aA]`.
Note that `output` preserves the mask dimensions `a1...aA`; this differs from `tf.boolean_mask`, which flattens those dimensions.

Args:

- **`data`**: A potentially ragged tensor.
- **`mask`**: A potentially ragged boolean tensor. `mask`'s shape must be a prefix of `data`'s shape.`rank(mask)` must be known statically.
- **`name`**: A name prefix for the returned tensor (optional).

Returns:
A potentially ragged tensor that is formed by retaining the elements in `data` where the corresponding value in `mask` is `True`.

- `rank(output) = rank(data)`.
- `output.ragged_rank = max(data.ragged_rank, rank(mask) - 1)`.

Raises:

- **`ValueError`**: if `rank(mask)` is not known statically; or if `mask.shape` is not a prefix of `data.shape`.

Examples:

```
>>> # Aliases for True & False so data and mask line up.
>>> T, F = (True, False)

>>> tf.ragged.boolean_mask(  # Mask a 2D Tensor.
...     data=[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
...     mask=[[T, F, T], [F, F, F], [T, F, F]]).tolist()
[[1, 3], [], [7]]

>>> tf.ragged.boolean_mask(  # Mask a 2D RaggedTensor.
...     tf.ragged.constant([[1, 2, 3], [4], [5, 6]]),
...     tf.ragged.constant([[F, F, T], [F], [T, T]])).tolist()
[[3], [], [5, 6]]

>>> tf.ragged.boolean_mask(  # Mask rows of a 2D RaggedTensor.
...     tf.ragged.constant([[1, 2, 3], [4], [5, 6]]),
...     tf.ragged.constant([True, False, True])).tolist()
[[1, 2, 3], [5, 6]]
```

tf.ragged.constant

- **Contents**
- Aliases:
- Used in the guide:

Constructs a constant RaggedTensor from a nested Python list.

Aliases:

- `tf.compat.v1.ragged.constant`
- `tf.compat.v2.ragged.constant`
- `tf.ragged.constant`

```
tf.ragged.constant(
    pylist,
    dtype=None,
    ragged_rank=None,
    inner_shape=None,
    name=None,
    row_splits_dtype=tf.dtypes.int64
)
```

Defined in `python/ops/ragged/ragged_factory_ops.py`.

Used in the guide:

- Ragged Tensors

Example:

```
>>> ragged.constant([[1, 2], [3], [4, 5, 6]]).eval()
RaggedTensorValue(values=[1, 2, 3, 4, 5, 6], splits=[0, 2, 3, 6])
```

All scalar values in `pylist` must have the same nesting depth `K`, and the returned `RaggedTensor`will have rank `K`. If `pylist` contains no scalar values, then `K` is one greater than the maximum depth of empty lists in `pylist`. All scalar values in `pylist` must be compatible with `dtype`.

Args:

- **`pylist`**: A nested `list`, `tuple` or `np.ndarray`. Any nested element that is not a `list`, `tuple` or `np.ndarray` must be a scalar value compatible with `dtype`.
- **`dtype`**: The type of elements for the returned `RaggedTensor`. If not specified, then a default is chosen based on the scalar values in `pylist`.
- **`ragged_rank`**: An integer specifying the ragged rank of the returned `RaggedTensor`. Must be nonnegative and less than `K`. Defaults to `max(0, K - 1)` if `inner_shape` is not specified. Defaults to `max(0, K
- 1 - len(inner_shape))`if`inner_shape` is specified.
- **`inner_shape`**: A tuple of integers specifying the shape for individual inner values in the returned `RaggedTensor`. Defaults to `()` if `ragged_rank` is not specified. If `ragged_rank` is specified, then a default is chosen based on the contents of `pylist`.
- **`name`**: A name prefix for the returned tensor (optional).
- **`row_splits_dtype`**: data type for the constructed `RaggedTensor`'s row_splits. One of `tf.int32` or `tf.int64`.

Returns:

A potentially ragged tensor with rank `K` and the specified `ragged_rank`, containing the values from `pylist`.

Raises:

- **`ValueError`**: If the scalar values in `pylist` have inconsistent nesting depth; or if ragged_rank or inner_shape are incompatible with `pylist`.

tf.ragged.map_flat_values

- **Contents**
- Aliases:
- Used in the guide:

Applies `op` to the values of one or more RaggedTensors.

Aliases:

- `tf.compat.v1.ragged.map_flat_values`
- `tf.compat.v2.ragged.map_flat_values`
- `tf.ragged.map_flat_values`

```
tf.ragged.map_flat_values(
    op,
    *args,
    **kwargs
)
```

Defined in `python/ops/ragged/ragged_functional_ops.py`.

Used in the guide:

- [Ragged Tensors]()

Replaces any `RaggedTensor` in `args` or `kwargs` with its `flat_values` tensor, and then calls `op`.

Returns a `RaggedTensor` that is constructed from the input `RaggedTensor`s' `nested_row_splits`and the value returned by the `op`.

If the input arguments contain multiple `RaggedTensor`s, then they must have identical `nested_row_splits`.

Examples:

```
>>> rt = ragged.constant([[1, 2, 3], [], [4, 5], [6]])
>>> ragged.map_flat_values(tf.ones_like, rt).eval().tolist()
[[1, 1, 1], [], [1, 1], [1]]
>>> ragged.map_flat_values(tf.multiply, rt, rt).eval().tolist()
[[1, 4, 9], [], [16, 25], [36]]
>>> ragged.map_flat_values(tf.add, rt, 5).eval().tolist()
[[6, 7, 8], [], [9, 10], [11]]
```

Args:

- **`op`**: The operation that should be applied to the RaggedTensor `flat_values`. `op` is typically an element-wise operation (such as math_ops.add), but any operation that preserves the size of the outermost dimension can be used. I.e., `shape[0]` of the value returned by `op` must match`shape[0]` of the `RaggedTensor`s' `flat_values` tensors.
- **`*args`**: Arguments for `op`.
- **`**kwargs`**: Keyword arguments for `op`.

Returns:

A `RaggedTensor` whose `ragged_rank` matches the `ragged_rank` of all input `RaggedTensor`s.

Raises:

- **`ValueError`**: If args contains no `RaggedTensors`, or if the `nested_splits` of the input `RaggedTensor`s are not identical.

tf.ragged.range

- **Contents**
- Aliases:

Returns a `RaggedTensor` containing the specified sequences of numbers.

Aliases:

- `tf.compat.v1.ragged.range`
- `tf.compat.v2.ragged.range`
- `tf.ragged.range`

```
tf.ragged.range(
    starts,
    limits=None,
    deltas=1,
    dtype=None,
    name=None,
    row_splits_dtype=tf.dtypes.int64
)
```

Defined in `python/ops/ragged/ragged_math_ops.py`.
Each row of the returned `RaggedTensor` contains a single sequence:

```
ragged.range(starts, limits, deltas)[i] ==
    tf.range(starts[i], limits[i], deltas[i])
```

If `start[i] < limits[i] and deltas[i] > 0`, then `output[i]` will be an empty list. Similarly, if `start[i] > limits[i] and deltas[i] < 0`, then `output[i]` will be an empty list. This behavior is consistent with the Python `range` function, but differs from the `tf.range` op, which returns an error for these cases.

Examples:

```
>>> ragged.range([3, 5, 2]).eval().tolist()
[[0, 1, 2], [0, 1, 2, 3, 4], [0, 1]]
>>> ragged.range([0, 5, 8], [3, 3, 12]).eval().tolist()
[[0, 1, 2], [], [8, 9, 10, 11]]
>>> ragged.range([0, 5, 8], [3, 3, 12], 2).eval().tolist()
[[0, 2], [], [8, 10]]
```

The input tensors `starts`, `limits`, and `deltas` may be scalars or vectors. The vector inputs must all have the same size. Scalar inputs are broadcast to match the size of the vector inputs.

Args:

- **`starts`**: Vector or scalar `Tensor`. Specifies the first entry for each range if `limits` is not `None`; otherwise, specifies the range limits, and the first entries default to `0`.
- **`limits`**: Vector or scalar `Tensor`. Specifies the exclusive upper limits for each range.
- **`deltas`**: Vector or scalar `Tensor`. Specifies the increment for each range. Defaults to `1`.
- **`dtype`**: The type of the elements of the resulting tensor. If not specified, then a value is chosen based on the other args.
- **`name`**: A name for the operation.
- **`row_splits_dtype`**: `dtype` for the returned `RaggedTensor`'s `row_splits` tensor. One of `tf.int32` or `tf.int64`.

Returns:

A `RaggedTensor` of type `dtype` with `ragged_rank=1`.

tf.ragged.row_splits_to_segment_ids

- **Contents**

- Aliases:

Generates the segmentation corresponding to a RaggedTensor `row_splits`.

Aliases:

- `tf.compat.v1.ragged.row_splits_to_segment_ids`
- `tf.compat.v2.ragged.row_splits_to_segment_ids`
- `tf.ragged.row_splits_to_segment_ids`

```
tf.ragged.row_splits_to_segment_ids(
    splits,
    name=None,
    out_type=None
)
```

Defined in `python/ops/ragged/segment_id_ops.py`.

Returns an integer vector `segment_ids`, where `segment_ids[i] == j` if `splits[j] <= i < splits[j+1]`. Example:

```
>>> ragged.row_splits_to_segment_ids([0, 3, 3, 5, 6, 9]).eval()
[ 0 0 0 2 2 3 4 4 4 ]
```

Args:

- **`splits`**: A sorted 1-D integer Tensor. `splits[0]` must be zero.
- **`name`**: A name prefix for the returned tensor (optional).
- **`out_type`**: The dtype for the return value. Defaults to `splits.dtype`, or `tf.int64` if `splits`does not have a dtype.

Returns:

A sorted 1-D integer Tensor, with `shape=[splits[-1]]`

Raises:

- **`ValueError`**: If `splits` is invalid.

tf.ragged.segment_ids_to_row_splits

- **Contents**
- Aliases:

Generates the RaggedTensor `row_splits` corresponding to a segmentation.

Aliases:

- `tf.compat.v1.ragged.segment_ids_to_row_splits`
- `tf.compat.v2.ragged.segment_ids_to_row_splits`
- `tf.ragged.segment_ids_to_row_splits`

```
tf.ragged.segment_ids_to_row_splits(
    segment_ids,
    num_segments=None,
    out_type=None,
    name=None
)
```

Defined in `python/ops/ragged/segment_id_ops.py`.

Returns an integer vector `splits`, where `splits[0] = 0` and `splits[i] = splits[i-1] + count(segment_ids==i)`. Example:

```
>>> ragged.segment_ids_to_row_splits([0, 0, 0, 2, 2, 3, 4, 4, 4]).eval()
[ 0 3 3 5 6 9 ]
```

Args:

- **`segment_ids`**: A 1-D integer Tensor.
- **`num_segments`**: A scalar integer indicating the number of segments. Defaults to `max(segment_ids) + 1` (or zero if `segment_ids` is empty).
- **`out_type`**: The dtype for the return value. Defaults to `segment_ids.dtype`, or `tf.int64` if `segment_ids` does not have a dtype.
- **`name`**: A name prefix for the returned tensor (optional).

Returns:
A sorted 1-D integer Tensor, with `shape=[num_segments + 1]`.

Module: tf.compat.v1.random / tf.random

- **Contents**
- Modules
- Functions

Public API for tf.random namespace.

Modules

`experimental` module: Public API for tf.random.experimental namespace.

Functions

`all_candidate_sampler(...)`: Generate the set of all classes.
`categorical(...)`: Draws samples from a categorical distribution.
`fixed_unigram_candidate_sampler(...)`: Samples a set of classes using the provided (fixed) base distribution.
`gamma(...)`: Draws `shape` samples from each of the given Gamma distribution(s).
`get_seed(...)`: Returns the local seeds an operation should use given an op-specific seed.
`learned_unigram_candidate_sampler(...)`: Samples a set of classes from a distribution learned during training.
`log_uniform_candidate_sampler(...)`: Samples a set of classes using a log-uniform (Zipfian) base distribution.
`multinomial(...)`: Draws samples from a multinomial distribution. (deprecated)
`normal(...)`: Outputs random values from a normal distribution.
`poisson(...)`: Draws `shape` samples from each of the given Poisson distribution(s).
`set_random_seed(...)`: Sets the graph-level random seed for the default graph.
`shuffle(...)`: Randomly shuffles a tensor along its first dimension.
`stateless_categorical(...)`: Draws deterministic pseudorandom samples from a categorical distribution.
`stateless_multinomial(...)`: Draws deterministic pseudorandom samples from a multinomial distribution. (deprecated)
`stateless_normal(...)`: Outputs deterministic pseudorandom values from a normal distribution.
`stateless_truncated_normal(...)`: Outputs deterministic pseudorandom values, truncated normally distributed.
`stateless_uniform(...)`: Outputs deterministic pseudorandom values from a uniform distribution.
`truncated_normal(...)`: Outputs random values from a truncated normal distribution.
`uniform(...)`: Outputs random values from a uniform distribution.
`uniform_candidate_sampler(...)`: Samples a set of classes using a uniform base distribution.

tf.compat.v1.random.stateless_multinomial

Draws deterministic pseudorandom samples from a multinomial distribution. (deprecated)

```
tf.compat.v1.random.stateless_multinomial(
    logits,
    num_samples,
    seed,
    output_dtype=tf.dtypes.int64,
    name=None
)
```

Defined in `python/ops/stateless_random_ops.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use **`tf.random.stateless_categorical`** instead.

This is a stateless version of `tf.random.categorical`: if run twice with the same seeds, it will produce the same pseudorandom numbers. The output is consistent across multiple runs on the same hardware (and between CPU and GPU), but may change between versions of TensorFlow or on non-CPU/GPU hardware.

Example:

```
# samples has shape [1, 5], where each value is either 0 or 1 with equal
# probability.
samples = tf.random.stateless_categorical(
    tf.math.log([[10., 10.]]), 5, seed=[7, 17])
```

Args:

- **`logits`**: 2-D Tensor with shape `[batch_size, num_classes]`. Each slice `[i, :]` represents the unnormalized log-probabilities for all classes.
- **`num_samples`**: 0-D. Number of independent samples to draw for each row slice.
- **`seed`**: A shape [2] integer Tensor of seeds to the random number generator.
- **`output_dtype`**: integer type to use for the output. Defaults to int64.
- **`name`**: Optional name for the operation.

Returns:

The drawn samples of shape `[batch_size, num_samples]`.

tf.random.all_candidate_sampler

- Contents
- Aliases:

Generate the set of all classes.

Aliases:

- `tf.compat.v1.nn.all_candidate_sampler`
- `tf.compat.v1.random.all_candidate_sampler`
- `tf.compat.v2.nn.all_candidate_sampler`
- `tf.compat.v2.random.all_candidate_sampler`
- `tf.nn.all_candidate_sampler`
- `tf.random.all_candidate_sampler`

```
tf.random.all_candidate_sampler(
    true_classes,
    num_true,
    num_sampled,
    unique,
```

```
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.
Deterministically generates and returns the set of all possible classes. For testing purposes. There is no need to use this, since you might as well use full softmax or full logistic regression.

Args:

- **`true_classes`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- **`num_true`**: An `int`. The number of target classes per training example.
- **`num_sampled`**: An `int`. The number of possible classes.
- **`unique`**: A `bool`. Ignored. unique.
- **`seed`**: An `int`. An operation-specific seed. Default is 0.
- **`name`**: A name for the operation (optional).

Returns:

- **`sampled_candidates`**: A tensor of type `int64` and shape `[num_sampled]`. This operation deterministically returns the entire range `[0, num_sampled]`.
- **`true_expected_count`**: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`. All returned values are 1.0.
- **`sampled_expected_count`**: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`. All returned values are 1.0.

tf.random.categorical

- Contents
- Aliases:
- Used in the tutorials:

Draws samples from a categorical distribution.

Aliases:

- `tf.compat.v1.random.categorical`
- `tf.compat.v2.random.categorical`
- `tf.random.categorical`

```
tf.random.categorical(
    logits,
    num_samples,
    dtype=None,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

Used in the tutorials:

- [Text generation with an RNN]()

Example:

```
# samples has shape [1, 5], where each value is either 0 or 1 with equal
# probability.
```

```
samples = tf.random.categorical(tf.math.log([[10., 10.]]), 5)
```

Args:

- **`logits`**: 2-D Tensor with shape `[batch_size, num_classes]`. Each slice `[i, :]` represents the unnormalized log-probabilities for all classes.
- **`num_samples`**: 0-D. Number of independent samples to draw for each row slice.
- **`dtype`**: integer type to use for the output. Defaults to int64.
- **`seed`**: A Python integer. Used to create a random seed for the distribution. See `tf.compat.v1.set_random_seed` for behavior.
- **`name`**: Optional name for the operation.

Returns:

The drawn samples of shape `[batch_size, num_samples]`.

tf.random.fixed_unigram_candidate_sampler

- Contents
- Aliases:

Samples a set of classes using the provided (fixed) base distribution.

Aliases:

- `tf.compat.v1.nn.fixed_unigram_candidate_sampler`
- `tf.compat.v1.random.fixed_unigram_candidate_sampler`
- `tf.compat.v2.nn.fixed_unigram_candidate_sampler`
- `tf.compat.v2.random.fixed_unigram_candidate_sampler`
- `tf.nn.fixed_unigram_candidate_sampler`
- `tf.random.fixed_unigram_candidate_sampler`

```
tf.random.fixed_unigram_candidate_sampler(
    true_classes,
    num_true,
    num_sampled,
    unique,
    range_max,
    vocab_file='',
    distortion=1.0,
    num_reserved_ids=0,
    num_shards=1,
    shard=0,
    unigrams=(),
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max)`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution is read from a file or passed in as an in-memory array. There is also an option to skew the distribution by applying a distortion power to the weights.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in [this document](). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- **`true_classes`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- **`num_true`**: An `int`. The number of target classes per training example.
- **`num_sampled`**: An `int`. The number of classes to randomly sample.
- **`unique`**: A `bool`. Determines whether all sampled classes in a batch are unique.
- **`range_max`**: An `int`. The number of possible classes.
- **`vocab_file`**: Each valid line in this file (which should have a CSV-like format) corresponds to a valid word ID. IDs are in sequential order, starting from num_reserved_ids. The last entry in each line is expected to be a value corresponding to the count or relative probability. Exactly one of `vocab_file` and `unigrams` needs to be passed to this operation.
- **`distortion`**: The distortion is used to skew the unigram probability distribution. Each weight is first raised to the distortion's power before adding to the internal unigram distribution. As a result,`distortion = 1.0` gives regular unigram sampling (as defined by the vocab file), and `distortion = 0.0` gives a uniform distribution.
- **`num_reserved_ids`**: Optionally some reserved IDs can be added in the range `[0, num_reserved_ids)` by the users. One use case is that a special unknown word token is used as ID 0. These IDs will have a sampling probability of 0.
- **`num_shards`**: A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `shard`) indicates the number of partitions that are being used in the overall computation.
- **`shard`**: A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `num_shards`) indicates the particular partition number of the operation, when partitioning is being used.
- **`unigrams`**: A list of unigram counts or probabilities, one per ID in sequential order. Exactly one of `vocab_file` and `unigrams` should be passed to this operation.
- **`seed`**: An `int`. An operation-specific seed. Default is 0.
- **`name`**: A name for the operation (optional).

Returns:

- **`sampled_candidates`**: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- **`true_expected_count`**: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- **`sampled_expected_count`**: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

tf.random.gamma

- Contents
- Aliases:

Draws `shape` samples from each of the given Gamma distribution(s).

Aliases:

- `tf.compat.v1.random.gamma`
- `tf.compat.v1.random_gamma`
- `tf.compat.v2.random.gamma`
- `tf.random.gamma`

```
tf.random.gamma(
    shape,
    alpha,
    beta=None,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

`alpha` is the shape parameter describing the distribution(s), and `beta` is the inverse scale parameter(s).

Note: Because internal calculations are done using **`float64`** and casting has **`floor`** semantics, we must manually map zero outcomes to the smallest possible positive floating-point value, i.e., **`np.finfo(dtype).tiny`**. This means that **`np.finfo(dtype).tiny`** occurs more frequently than it otherwise should. This bias can only happen for small values of **`alpha`**, i.e., **`alpha << 1`** or large values of **`beta`**, i.e., **`beta >> 1`**.

The samples are differentiable w.r.t. alpha and beta. The derivatives are computed using the approach described in the paper

Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Example:

```
samples = tf.random.gamma([10], [0.5, 1.5])
# samples has shape [10, 2], where each slice [:, 0] and [:, 1] represents
# the samples drawn from each distribution

samples = tf.random.gamma([7, 5], [0.5, 1.5])
# samples has shape [7, 5, 2], where each slice [:, :, 0] and [:, :, 1]
# represents the 7x5 samples drawn from each of the two distributions

alpha = tf.constant([[1.],[3.],[5.]])
beta = tf.constant([[3., 4.]])
samples = tf.random.gamma([30], alpha=alpha, beta=beta)
# samples has shape [30, 3, 2], with 30 samples each of 3x2 distributions.

loss = tf.reduce_mean(tf.square(samples))
dloss_dalpha, dloss_dbeta = tf.gradients(loss, [alpha, beta])
# unbiased stochastic derivatives of the loss function
alpha.shape == dloss_dalpha.shape  # True
beta.shape == dloss_dbeta.shape  # True
```

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output samples to be drawn per alpha/beta-parameterized distribution.
- **`alpha`**: A Tensor or Python value or N-D array of type `dtype`. `alpha` provides the shape parameter(s) describing the gamma distribution(s) to sample. Must be broadcastable with `beta`.
- **`beta`**: A Tensor or Python value or N-D array of type `dtype`. Defaults to 1. `beta` provides the inverse scale parameter(s) of the gamma distribution(s) to sample. Must be broadcastable with `alpha`.
- **`dtype`**: The type of alpha, beta, and the output: `float16`, `float32`, or `float64`.

- **`seed`**: A Python integer. Used to create a random seed for the distributions. See`tf.compat.v1.set_random_seed` for behavior.
- **`name`**: Optional name for the operation.

Returns:

- **`samples`**: a `Tensor` of shape `tf.concat([shape, tf.shape(alpha + beta)], axis=0)` with values of type `dtype`.

tf.random.learned_unigram_candidate_sampler

- Contents
- Aliases:

Samples a set of classes from a distribution learned during training.

Aliases:

- `tf.compat.v1.nn.learned_unigram_candidate_sampler`
- `tf.compat.v1.random.learned_unigram_candidate_sampler`
- `tf.compat.v2.nn.learned_unigram_candidate_sampler`
- `tf.compat.v2.random.learned_unigram_candidate_sampler`
- `tf.nn.learned_unigram_candidate_sampler`
- `tf.random.learned_unigram_candidate_sampler`

```
tf.random.learned_unigram_candidate_sampler(
    true_classes,
    num_true,
    num_sampled,
    unique,
    range_max,
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max)`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is constructed on the fly during training. It is a unigram distribution over the target classes seen so far during training. Every integer in `[0, range_max)` begins with a weight of 1, and is incremented by 1 each time it is seen as a target class. The base distribution is not saved to checkpoints, so it is reset when the model is reloaded.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in [this document](). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- **`true_classes`**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- **`num_true`**: An `int`. The number of target classes per training example.
- **`num_sampled`**: An `int`. The number of classes to randomly sample.
- **`unique`**: A `bool`. Determines whether all sampled classes in a batch are unique.
- **`range_max`**: An `int`. The number of possible classes.

- **seed**: An `int`. An operation-specific seed. Default is 0.
- **name**: A name for the operation (optional).

Returns:

- **sampled_candidates**: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- **true_expected_count**: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- **sampled_expected_count**: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

tf.random.log_uniform_candidate_sampler

- Contents
- Aliases:

Samples a set of classes using a log-uniform (Zipfian) base distribution.

Aliases:

- `tf.compat.v1.nn.log_uniform_candidate_sampler`
- `tf.compat.v1.random.log_uniform_candidate_sampler`
- `tf.compat.v2.random.log_uniform_candidate_sampler`
- `tf.random.log_uniform_candidate_sampler`

```
tf.random.log_uniform_candidate_sampler(
    true_classes,
    num_true,
    num_sampled,
    unique,
    range_max,
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max)`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is an approximately log-uniform or Zipfian distribution:

`P(class) = (log(class + 2) - log(class + 1)) / log(range_max + 1)`

This sampler is useful when the target classes approximately follow such a distribution - for example, if the classes represent words in a lexicon sorted in decreasing order of frequency. If your classes are not ordered by decreasing frequency, do not use this op.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in this document. If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- **true_classes**: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- **num_true**: An `int`. The number of target classes per training example.
- **num_sampled**: An `int`. The number of classes to randomly sample.
- **unique**: A `bool`. Determines whether all sampled classes in a batch are unique.

- **range_max**: An `int`. The number of possible classes.
- **seed**: An `int`. An operation-specific seed. Default is 0.
- **name**: A name for the operation (optional).

Returns:

- **sampled_candidates**: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- **true_expected_count**: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- **sampled_expected_count**: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

tf.random.normal

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Outputs random values from a normal distribution.

Aliases:

- `tf.compat.v1.random.normal`
- `tf.compat.v1.random_normal`
- `tf.compat.v2.random.normal`
- `tf.random.normal`

```
tf.random.normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

Used in the guide:

- Eager essentials

Used in the tutorials:

- Convolutional Variational Autoencoder
- Custom training: basics
- Deep Convolutional Generative Adversarial Network

Args:

- **shape**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean**: A 0-D Tensor or Python value of type `dtype`. The mean of the normal distribution.
- **stddev**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution.
- **dtype**: The type of the output.
- **seed**: A Python integer. Used to create a random seed for the distribution. See`tf.compat.v1.set_random_seed` for behavior.
- **name**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random normal values.

tf.random.poisson

- Contents
- Aliases:

Draws `shape` samples from each of the given Poisson distribution(s).

Aliases:

- `tf.compat.v2.random.poisson`
- `tf.random.poisson`

```
tf.random.poisson(
    shape,
    lam,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.
`lam` is the rate parameter describing the distribution(s).

Example:

```
samples = tf.random.poisson([10], [0.5, 1.5])
# samples has shape [10, 2], where each slice [:, 0] and [:, 1] represents
# the samples drawn from each distribution

samples = tf.random.poisson([7, 5], [12.2, 3.3])
# samples has shape [7, 5, 2], where each slice [:, :, 0] and [:, :, 1]
# represents the 7x5 samples drawn from each of the two distributions
```

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output samples to be drawn per "rate"-parameterized distribution.
- **`lam`**: A Tensor or Python value or N-D array of type `dtype`. `lam` provides the rate parameter(s) describing the poisson distribution(s) to sample.
- **`dtype`**: The type of the output: `float16`, `float32`, `float64`, `int32` or `int64`.
- **`seed`**: A Python integer. Used to create a random seed for the distributions. See`tf.compat.v1.set_random_seed` for behavior.
- **`name`**: Optional name for the operation.

Returns:

- **`samples`**: a `Tensor` of shape `tf.concat([shape, tf.shape(lam)], axis=0)` with values of type `dtype`.

tf.random.set_seed

- Contents
- Aliases:

Sets the graph-level random seed.

Aliases:

- `tf.compat.v2.random.set_seed`
- `tf.random.set_seed`

```
tf.random.set_seed(seed)
```

Defined in `python/framework/random_seed.py`.

Operations that rely on a random seed actually derive it from two seeds: the graph-level and operation-level seeds. This sets the graph-level seed.

Its interactions with operation-level seeds is as follows:

1. If neither the graph-level nor the operation seed is set: A random seed is used for this op.
2. If the graph-level seed is set, but the operation seed is not: The system deterministically picks an operation seed in conjunction with the graph-level seed so that it gets a unique random sequence.
3. If the graph-level seed is not set, but the operation seed is set: A default graph-level seed and the specified operation seed are used to determine the random sequence.
4. If both the graph-level and the operation seed are set: Both seeds are used in conjunction to determine the random sequence.

To illustrate the user-visible effects, consider these examples:

To generate different sequences across sessions, set neither graph-level nor op-level seeds:

```
a = tf.random.uniform([1])
b = tf.random.normal([1])

print("Session 1")
with tf.compat.v1.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
with tf.compat.v1.Session() as sess2:
  print(sess2.run(a))  # generates 'A3'
  print(sess2.run(a))  # generates 'A4'
  print(sess2.run(b))  # generates 'B3'
  print(sess2.run(b))  # generates 'B4'
```

To generate the same repeatable sequence for an op across sessions, set the seed for the op:

```
a = tf.random.uniform([1], seed=1)
b = tf.random.normal([1])

# Repeatedly running this block with the same graph will generate the same
# sequence of values for 'a', but different sequences of values for 'b'.
print("Session 1")
with tf.compat.v1.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
```

```
with tf.compat.v1.Session() as sess2:
  print(sess2.run(a))  # generates 'A1'
  print(sess2.run(a))  # generates 'A2'
  print(sess2.run(b))  # generates 'B3'
  print(sess2.run(b))  # generates 'B4'
```

To make the random sequences generated by all ops be repeatable across sessions, set a graph-level seed:

```
tf.random.set_seed(1234)
a = tf.random.uniform([1])
b = tf.random.normal([1])

# Repeatedly running this block with the same graph will generate the same
# sequences of 'a' and 'b'.
print("Session 1")
with tf.compat.v1.Session() as sess1:
  print(sess1.run(a))  # generates 'A1'
  print(sess1.run(a))  # generates 'A2'
  print(sess1.run(b))  # generates 'B1'
  print(sess1.run(b))  # generates 'B2'

print("Session 2")
with tf.compat.v1.Session() as sess2:
  print(sess2.run(a))  # generates 'A1'
  print(sess2.run(a))  # generates 'A2'
  print(sess2.run(b))  # generates 'B1'
  print(sess2.run(b))  # generates 'B2'
```

Args:

- `seed`: integer.

tf.random.shuffle

- Contents
- Aliases:

Randomly shuffles a tensor along its first dimension.

Aliases:

- `tf.compat.v1.random.shuffle`
- `tf.compat.v1.random_shuffle`
- `tf.compat.v2.random.shuffle`
- `tf.random.shuffle`

```
tf.random.shuffle(
    value,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

The tensor is shuffled along dimension 0, such that each `value[j]` is mapped to one and only one `output[i]`. For example, a mapping that might occur for a 3x2 tensor is:

```
[[1, 2],       [[5, 6],
 [3, 4],  ==>   [1, 2],
 [5, 6]]        [3, 4]]
```

Args:

- **`value`**: A Tensor to be shuffled.
- **`seed`**: A Python integer. Used to create a random seed for the distribution. See`tf.compat.v1.set_random_seed` for behavior.
- **`name`**: A name for the operation (optional).

Returns:
A tensor of same shape and type as `value`, shuffled along its first dimension.

tf.random.stateless_categorical

- Contents
- Aliases:

Draws deterministic pseudorandom samples from a categorical distribution.

Aliases:

- `tf.compat.v1.random.stateless_categorical`
- `tf.compat.v2.random.stateless_categorical`
- `tf.random.stateless_categorical`

```
tf.random.stateless_categorical(
    logits,
    num_samples,
    seed,
    dtype=tf.dtypes.int64,
    name=None
)
```

Defined in `python/ops/stateless_random_ops.py`.
This is a stateless version of `tf.categorical`: if run twice with the same seeds, it will produce the same pseudorandom numbers. The output is consistent across multiple runs on the same hardware (and between CPU and GPU), but may change between versions of TensorFlow or on non-CPU/GPU hardware.

Example:

```
# samples has shape [1, 5], where each value is either 0 or 1 with equal
# probability.
samples = tf.random.stateless_categorical(
    tf.math.log([[10., 10.]]), 5, seed=[7, 17])
```

Args:

- **`logits`**: 2-D Tensor with shape `[batch_size, num_classes]`. Each slice `[i, :]` represents the unnormalized log-probabilities for all classes.
- **`num_samples`**: 0-D. Number of independent samples to draw for each row slice.
- **`seed`**: A shape [2] integer Tensor of seeds to the random number generator.
- **`dtype`**: integer type to use for the output. Defaults to int64.
- **`name`**: Optional name for the operation.

Returns:
The drawn samples of shape `[batch_size, num_samples]`.

tf.random.stateless_normal

- Contents
- Aliases:

Outputs deterministic pseudorandom values from a normal distribution.

Aliases:

- `tf.compat.v1.random.stateless_normal`
- `tf.compat.v2.random.stateless_normal`
- `tf.random.stateless_normal`

```
tf.random.stateless_normal(
    shape,
    seed,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in `python/ops/stateless_random_ops.py`.
This is a stateless version of `tf.random.normal`: if run twice with the same seeds, it will produce the same pseudorandom numbers. The output is consistent across multiple runs on the same hardware (and between CPU and GPU), but may change between versions of TensorFlow or on non-CPU/GPU hardware.

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`seed`**: A shape [2] integer Tensor of seeds to the random number generator.
- **`mean`**: A 0-D Tensor or Python value of type `dtype`. The mean of the normal distribution.
- **`stddev`**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution.
- **`dtype`**: The type of the output.
- **`name`**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random normal values.

tf.random.stateless_truncated_normal

- Contents
- Aliases:

Outputs deterministic pseudorandom values, truncated normally distributed.

Aliases:

- `tf.compat.v1.random.stateless_truncated_normal`
- `tf.compat.v2.random.stateless_truncated_normal`
- `tf.random.stateless_truncated_normal`

```
tf.random.stateless_truncated_normal(
    shape,
    seed,
```

```
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in python/ops/stateless_random_ops.py.
This is a stateless version of tf.random.truncated_normal: if run twice with the same seeds, it will produce the same pseudorandom numbers. The output is consistent across multiple runs on the same hardware (and between CPU and GPU), but may change between versions of TensorFlow or on non-CPU/GPU hardware.
The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

Args:

- **shape**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **seed**: A shape [2] integer Tensor of seeds to the random number generator.
- **mean**: A 0-D Tensor or Python value of type `dtype`. The mean of the truncated normal distribution.
- **stddev**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution, before truncation.
- **dtype**: The type of the output.
- **name**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random truncated normal values.

tf.random.stateless_truncated_normal

- Contents
- Aliases:

Outputs deterministic pseudorandom values, truncated normally distributed.

Aliases:

- `tf.compat.v1.random.stateless_truncated_normal`
- `tf.compat.v2.random.stateless_truncated_normal`
- `tf.random.stateless_truncated_normal`

```
tf.random.stateless_truncated_normal(
    shape,
    seed,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in python/ops/stateless_random_ops.py.
This is a stateless version of tf.random.truncated_normal: if run twice with the same seeds, it will produce the same pseudorandom numbers. The output is consistent across multiple runs on the same hardware (and between CPU and GPU), but may change between versions of TensorFlow or on non-CPU/GPU hardware.

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

Args:

- **shape**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **seed**: A shape [2] integer Tensor of seeds to the random number generator.
- **mean**: A 0-D Tensor or Python value of type `dtype`. The mean of the truncated normal distribution.
- **stddev**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution, before truncation.
- **dtype**: The type of the output.
- **name**: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random truncated normal values.

tf.random.truncated_normal

- Contents
- Aliases:
- Used in the guide:

Outputs random values from a truncated normal distribution.

Aliases:

- `tf.compat.v1.random.truncated_normal`
- `tf.compat.v1.truncated_normal`
- `tf.compat.v2.random.truncated_normal`
- `tf.random.truncated_normal`

```
tf.random.truncated_normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

Used in the guide:

- Ragged Tensors

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

Args:

- **shape**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean**: A 0-D Tensor or Python value of type `dtype`. The mean of the truncated normal distribution.
- **stddev**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution, before truncation.
- **dtype**: The type of the output.
- **seed**: A Python integer. Used to create a random seed for the distribution. See`tf.compat.v1.set_random_seed` for behavior.
- **name**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random truncated normal values
.

tf.random.uniform

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Outputs random values from a uniform distribution.

Aliases:

- `tf.compat.v1.random.uniform`
- `tf.compat.v1.random_uniform`
- `tf.compat.v2.random.uniform`
- `tf.random.uniform`

```
tf.random.uniform(
    shape,
    minval=0,
    maxval=None,
    dtype=tf.dtypes.float32,
    seed=None,
    name=None
)
```

Defined in `python/ops/random_ops.py`.

Used in the guide:

- tf.function and AutoGraph in TensorFlow 2.0

Used in the tutorials:

- Neural Machine Translation with Attention
- Pix2Pix
- Tensors and Operations
- Transformer model for language understanding
- tf.function

The generated values follow a uniform distribution in the range `[minval, maxval)`. The lower bound `minval` is included in the range, while the upper bound `maxval` is excluded.
For floats, the default range is `[0, 1)`. For ints, at least `maxval` must be specified explicitly.
In the integer case, the random integers are slightly biased unless `maxval - minval` is an exact power of two. The bias is small for values of `maxval - minval` significantly smaller than the range of the output (either `2**32` or `2**64`).

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`minval`**: A 0-D Tensor or Python value of type `dtype`. The lower bound on the range of random values to generate. Defaults to 0.
- **`maxval`**: A 0-D Tensor or Python value of type `dtype`. The upper bound on the range of random values to generate. Defaults to 1 if `dtype` is floating point.
- **`dtype`**: The type of the output: `float16`, `float32`, `float64`, `int32`, or `int64`.

- `seed`: A Python integer. Used to create a random seed for the distribution. See `tf.compat.v1.set_random_seed` for behavior.
- `name`: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random uniform values.

Raises:

- `ValueError`: If `dtype` is integral and `maxval` is not specified.

tf.random.uniform_candidate_sampler

- Contents
- Aliases:

Samples a set of classes using a uniform base distribution.

Aliases:

- `tf.compat.v1.nn.uniform_candidate_sampler`
- `tf.compat.v1.random.uniform_candidate_sampler`
- `tf.compat.v2.random.uniform_candidate_sampler`
- `tf.random.uniform_candidate_sampler`

```
tf.random.uniform_candidate_sampler(
    true_classes,
    num_true,
    num_sampled,
    unique,
    range_max,
    seed=None,
    name=None
)
```

Defined in `python/ops/candidate_sampling_ops.py`.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max)`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is the uniform distribution over the range of integers `[0, range_max)`.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count`representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to `Q(y|x)` defined in this document. If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true`: An `int`. The number of target classes per training example.
- `num_sampled`: An `int`. The number of classes to randomly sample. The `sampled_candidates`return value will have shape `[num_sampled]`. If `unique=True`, `num_sampled` must be less than or equal to `range_max`.
- `unique`: A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max`: An `int`. The number of possible classes.

- **`seed`**: An `int`. An operation-specific seed. Default is 0.
- **`name`**: A name for the operation (optional).

Returns:

- **`sampled_candidates`**: A tensor of type `int64` and shape `[num_sampled]`. The sampled classes, either with possible duplicates (`unique=False`) or all unique (`unique=True`). In either case, `sampled_candidates` is independent of the true classes.
- **`true_expected_count`**: A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- **`sampled_expected_count`**: A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

Module: tf.random.experimental

- **Contents**
- Classes
- Functions

Public API for tf.random.experimental namespace.

Classes

`class Generator`: Random-number generator.

Functions

`create_rng_state(...)`: Creates a RNG state.
`get_global_generator(...)`
`set_global_generator(...)`: Replaces the global generator with another `Generator` object.

tf.random.experimental.create_rng_state

- Contents
- Aliases:

Creates a RNG state.

Aliases:

- `tf.compat.v1.random.experimental.create_rng_state`
- `tf.compat.v2.random.experimental.create_rng_state`
- `tf.random.experimental.create_rng_state`

```
tf.random.experimental.create_rng_state(
    seed,
    algorithm
)
```

Defined in `python/ops/stateful_random_ops.py`.

Args:

- **`seed`**: an integer or 1-D tensor.
- **`algorithm`**: an integer representing the RNG algorithm.

Returns:

a 1-D tensor whose size depends on the algorithm.

tf.random.experimental.Generator

- Contents
- Class Generator
  - Aliases:

- __init__
- Properties

Class `Generator`

Random-number generator.

Aliases:

- Class `tf.compat.v1.random.experimental.Generator`
- Class `tf.compat.v2.random.experimental.Generator`
- Class `tf.random.experimental.Generator`

Defined in `python/ops/stateful_random_ops.py`.

It uses Variable to manage its internal state, and allows choosing an Random-Number-Generation (RNG) algorithm.

CPU, GPU and TPU with the same algorithm and seed will generate the same integer random numbers. Float-point results (such as the output of `normal`) may have small numerical discrepancies between CPU and GPU.

`__init__`

```
__init__(
    copy_from=None,
    state=None,
    alg=None
)
```

Creates a generator.

The new generator will be initialized by one of the following ways, with decreasing precedence: (1) If `copy_from` is not None, the new generator is initialized by copying information from another generator. (3) If `state` and `alg` are not None (they must be set together), the new generator is initialized by a state.

Args:

- **`copy_from`**: a generator to be copied from.
- **`state`**: a vector of dtype STATE_TYPE representing the initial state of the RNG, whose length and semantics are algorithm-specific.
- **`alg`**: the RNG algorithm. Possible values are RNG_ALG_PHILOX for the Philox algorithm and RNG_ALG_THREEFRY for the ThreeFry algorithm (see paper 'Parallel Random Numbers: As Easy as 1, 2, 3' [https://www.thesalmons.org/john/random123/papers/random123sc11.pdf]).

Properties

`algorithm`

The RNG algorithm.

`key`

The 'key' part of the state of a counter-based RNG.

For a counter-base RNG algorithm such as Philox and ThreeFry (as described in paper 'Parallel Random Numbers: As Easy as 1, 2, 3' [https://www.thesalmons.org/john/random123/papers/random123sc11.pdf]), the RNG state consists of two parts: counter and key. The output is generated via the formula: output=hash(key, counter), i.e. a hashing of the counter parametrized by the key. Two RNGs with two different keys can be thought as generating two independent random-number streams (a stream is formed by increasing the counter).

Returns:
A scalar which is the 'key' part of the state, if the RNG algorithm is counter-based; otherwise it raises a ValueError.

`state`
The internal state of the RNG.

Methods

`binomial`

```
binomial(
    shape,
    counts,
    probs,
    dtype=tf.dtypes.int32,
    name=None
)
```

Outputs random values from a binomial distribution.
The generated values follow a binomial distribution with specified count and probability of success parameters.

Example:

```
counts = [10., 20.]
# Probability of success.
probs = [0.8, 0.9]

rng = tf.random.experimental.Generator(seed=234)
binomial_samples = rng.binomial(shape=[2], counts=counts, probs=probs)
```

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`counts`**: A 0/1-D Tensor or Python value`. The counts of the binomial distribution.
- **`probs`**: A 0/1-D Tensor or Python value`. The probability of success for the binomial distribution.
- **`dtype`**: The type of the output. Default: tf.int32
- **`name`**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random binomial values.

`from_key_counter`

```
@classmethod
from_key_counter(
    cls,
    key,
    counter,
    alg
)
```

Creates a generator from a key and a counter.
This constructor only applies if the algorithm is a counter-based algorithm. See method `key` for the meaning of "key" and "counter".

Args:

- **`key`**: the key for the RNG, a scalar of type STATE_TYPE.
- **`counter`**: a vector of dtype STATE_TYPE representing the initial counter for the RNG, whose length is algorithm-specific.,
- **`alg`**: the RNG algorithm. If None, it will be auto-selected. See `__init__` for its possible values.

Returns:
The new generator.

from_non_deterministic_state

```
@classmethod
from_non_deterministic_state(
    cls,
    alg=None
)
```

Creates a generator by non-deterministically initializing its state.
The source of the non-determinism will be platform- and time-dependent.

Args:

- **`alg`**: (optional) the RNG algorithm. If None, it will be auto-selected. See `__init__` for its possible values.

Returns:
The new generator.

from_seed

```
@classmethod
from_seed(
    cls,
    seed,
    alg=None
)
```

Creates a generator from a seed.
A seed is a 1024-bit unsigned integer represented either as a Python integer or a vector of integers. Seeds shorter than 1024-bit will be padded. The padding, the internal structure of a seed and the way a seed is converted to a state are all opaque (unspecified). The only semantics specification of seeds is that two different seeds are likely to produce two independent generators (but no guarantee).

Args:

- **`seed`**: the seed for the RNG.
- **`alg`**: (optional) the RNG algorithm. If None, it will be auto-selected. See `__init__` for its possible values.

Returns:
The new generator.

from_state

```
@classmethod
from_state(
    cls,
    state,
    alg
```

```
)
```

Creates a generator from a state.
See `__init__` for description of `state` and `alg`.

Args:

- **`state`**: the new state.
- **`alg`**: the RNG algorithm.

Returns:
The new generator.

make_seeds

```
make_seeds(count=1)
```

Generates seeds for stateless random ops.

For example:

```
seeds = get_global_generator().make_seeds(count=10)
for i in range(10):
  seed = seeds[:, i]
  numbers = stateless_random_normal(shape=[2, 3], seed=seed)
  ...
```

Args:

- **`count`**: the number of seed pairs (note that stateless random ops need a pair of seeds to invoke).

Returns:
A tensor of shape [2, count] and dtype int64.

normal

```
normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Outputs random values from a normal distribution.

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`mean`**: A 0-D Tensor or Python value of type `dtype`. The mean of the normal distribution.
- **`stddev`**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution.
- **`dtype`**: The type of the output.
- **`name`**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random normal values.

reset

```
reset(state)
```

Resets the generator by a new state.
See `__init__` for the meaning of "state".

Args:

- **`state`**: the new state.

reset_from_key_counter

```
reset_from_key_counter(
    key,
    counter
)
```

Resets the generator by a new key-counter pair.
See `from_key_counter` for the meaning of "key" and "counter".

Args:

- **`key`**: the new key.
- **`counter`**: the new counter.

reset_from_seed

```
reset_from_seed(seed)
```

Resets the generator by a new seed.
See `from_seed` for the meaning of "seed".

Args:

- **`seed`**: the new seed.

skip

```
skip(delta)
```

Advance the counter of a counter-based RNG.

Args:

- **`delta`**: the amount of advancement. The state of the RNG after `skip(n)` will be the same as that after `normal([n])` (or any other distribution). The actual increment added to the counter is an unspecified implementation detail.

split

```
split(count=1)
```

Returns a list of independent `Generator` objects.
Two generators are independent of each other in the sense that the random-number streams they generate don't have statistically detectable correlations. The new generators are also independent of the old one. The old generator's state will be changed (like other random-number generating methods), so two calls of `split` will return different new generators.

For example:

```
gens = get_global_generator().split(count=10)
for gen in gens:
  numbers = gen.normal(shape=[2, 3])
  # ...
gens2 = get_global_generator().split(count=10)
# gens2 will be different from gens
```

The new generators will be put on the current device (possible different from the old generator's), for example:

```
with tf.device("/device:CPU:0"):
  gen = Generator(seed=1234)  # gen is on CPU
with tf.device("/device:GPU:0"):
  gens = gen.split(count=10)  # gens are on GPU
```

Args:

- **`count`**: the number of generators to return.

Returns:

A list (length `count`) of `Generator` objects independent of each other. The new generators have the same RNG algorithm as the old one.

`truncated_normal`

```
truncated_normal(
    shape,
    mean=0.0,
    stddev=1.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Outputs random values from a truncated normal distribution.

The generated values follow a normal distribution with specified mean and standard deviation, except that values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked.

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`mean`**: A 0-D Tensor or Python value of type `dtype`. The mean of the truncated normal distribution.
- **`stddev`**: A 0-D Tensor or Python value of type `dtype`. The standard deviation of the normal distribution, before truncation.
- **`dtype`**: The type of the output.
- **`name`**: A name for the operation (optional).

Returns:

A tensor of the specified shape filled with random truncated normal values.

`uniform`

```
uniform(
    shape,
    minval=0,
    maxval=None,
    dtype=tf.dtypes.float32,
    name=None
)
```

Outputs random values from a uniform distribution.

The generated values follow a uniform distribution in the range `[minval, maxval)`. The lower bound `minval` is included in the range, while the upper bound `maxval` is excluded. (For float

numbers especially low-precision types like bfloat16, because of rounding, the result may sometimes include `maxval`.)
For floats, the default range is `[0, 1)`. For ints, at least `maxval` must be specified explicitly.
In the integer case, the random integers are slightly biased unless `maxval - minval` is an exact power of two. The bias is small for values of `maxval - minval` significantly smaller than the range of the output (either `2**32` or `2**64`).

Args:

- **`shape`**: A 1-D integer Tensor or Python array. The shape of the output tensor.
- **`minval`**: A 0-D Tensor or Python value of type `dtype`. The lower bound on the range of random values to generate. Defaults to 0.
- **`maxval`**: A 0-D Tensor or Python value of type `dtype`. The upper bound on the range of random values to generate. Defaults to 1 if `dtype` is floating point.
- **`dtype`**: The type of the output.
- **`name`**: A name for the operation (optional).

Returns:
A tensor of the specified shape filled with random uniform values.

Raises:

- **`ValueError`**: If `dtype` is integral and `maxval` is not specified.

`uniform_full_int`

```
uniform_full_int(
    shape,
    dtype=tf.dtypes.uint64,
    name=None
)
```

Uniform distribution on an integer type's entire range.
The other method `uniform` only covers the range [minval, maxval), which cannot be `dtype`'s full range because `maxval` is of type `dtype`.

Args:

- **`shape`**: the shape of the output.
- **`dtype`**: (optional) the integer type, default to uint64.
- **`name`**: (optional) the name of the node.

Returns:
A tensor of random numbers of the required shape.

tf.random.experimental.get_global_generator

- Contents
- Aliases:

Aliases:

- `tf.compat.v1.random.experimental.get_global_generator`
- `tf.compat.v2.random.experimental.get_global_generator`
- `tf.random.experimental.get_global_generator`

```
tf.random.experimental.get_global_generator()
```

Defined in `python/ops/stateful_random_ops.py`.

tf.random.experimental.set_global_generator

- Contents
- Aliases:

Replaces the global generator with another `Generator` object.

Aliases:

- `tf.compat.v1.random.experimental.set_global_generator`
- `tf.compat.v2.random.experimental.set_global_generator`
- `tf.random.experimental.set_global_generator`

```
tf.random.experimental.set_global_generator(generator)
```

Defined in `python/ops/stateful_random_ops.py`.

This function creates a new Generator object (and the Variable object within), which does not work well with tf.function because (1) tf.function puts restrictions on Variable creation thus reset_global_generator can't be freely used inside tf.function; (2) redirecting a global variable to a new object is problematic with tf.function because the old object may be captured by a 'tf.function'ed function and still be used by it. A 'tf.function'ed function only keeps weak references to variables, so deleting a variable and then calling that function again may raise an error, as demonstrated by random_test.py/RandomTest.testResetGlobalGeneratorBadWithDefun .

Args:

- **`generator`**: the new `Generator` object.

Module: tf.compat.v1.saved_model / tf.saved_model

- **Contents**
- Modules
- Classes
- Functions
- Other Members

Public API for tf.saved_model namespace.

Modules

`builder` module: SavedModel builder.

`constants` module: Constants for SavedModel save and restore operations.

`experimental` module: Public API for tf.saved_model.experimental namespace.

`loader` module: Loader functionality for SavedModel with hermetic, language-neutral exports.

`main_op` module: SavedModel main op.

`signature_constants` module: Signature constants for SavedModel save and restore operations.

`signature_def_utils` module: SignatureDef utility functions.

`tag_constants` module: Common tags used for graphs in SavedModel.

`utils` module: SavedModel utility functions.

Classes

`class Builder`: Builds the `SavedModel` protocol buffer and saves variables and assets.

Functions

`build_signature_def(...)`: Utility function to build a SignatureDef protocol buffer.

`build_tensor_info(...)`: Utility function to build TensorInfo proto from a Tensor. (deprecated)

`classification_signature_def(...)`: Creates classification signature from given examples and predictions.

`contains_saved_model(...)`: Checks whether the provided export directory could contain a SavedModel.

`get_tensor_from_tensor_info(...)`: Returns the Tensor or SparseTensor described by a TensorInfo proto. (deprecated)

`is_valid_signature(...)`: Determine whether a SignatureDef can be served by TensorFlow Serving.
`load(...)`: Loads the model from a SavedModel as specified by tags. (deprecated)
`load_v2(...)`: Load a SavedModel from `export_dir`.
`main_op_with_restore(...)`: Returns a main op to init variables, tables and restore the graph. (deprecated)
`maybe_saved_model_directory(...)`: Checks whether the provided export directory could contain a SavedModel.
`predict_signature_def(...)`: Creates prediction signature from given inputs and outputs.
`regression_signature_def(...)`: Creates regression signature from given examples and predictions.
`save(...)`: Exports the Trackable object `obj` to SavedModel format.
`simple_save(...)`: Convenience function to build a SavedModel suitable for serving. (deprecated)

Other Members

- `ASSETS_DIRECTORY = 'assets'`
- `ASSETS_KEY = 'saved_model_assets'`
- `CLASSIFY_INPUTS = 'inputs'`
- `CLASSIFY_METHOD_NAME = 'tensorflow/serving/classify'`
- `CLASSIFY_OUTPUT_CLASSES = 'classes'`
- `CLASSIFY_OUTPUT_SCORES = 'scores'`
- `DEFAULT_SERVING_SIGNATURE_DEF_KEY = 'serving_default'`
- `GPU = 'gpu'`
- `LEGACY_INIT_OP_KEY = 'legacy_init_op'`
- `MAIN_OP_KEY = 'saved_model_main_op'`
- `PREDICT_INPUTS = 'inputs'`
- `PREDICT_METHOD_NAME = 'tensorflow/serving/predict'`
- `PREDICT_OUTPUTS = 'outputs'`
- `REGRESS_INPUTS = 'inputs'`
- `REGRESS_METHOD_NAME = 'tensorflow/serving/regress'`
- `REGRESS_OUTPUTS = 'outputs'`
- `SAVED_MODEL_FILENAME_PB = 'saved_model.pb'`
- `SAVED_MODEL_FILENAME_PBTXT = 'saved_model.pbtxt'`
- `SAVED_MODEL_SCHEMA_VERSION = 1`
- `SERVING = 'serve'`
- `TPU = 'tpu'`
- `TRAINING = 'train'`
- `VARIABLES_DIRECTORY = 'variables'`
- `VARIABLES_FILENAME = 'variables'`

tf.saved_model.contains_saved_model

- Contents
- Aliases:

Checks whether the provided export directory could contain a SavedModel.

Aliases:

- `tf.compat.v2.saved_model.contains_saved_model`
- `tf.saved_model.contains_saved_model`

```
tf.saved_model.contains_saved_model(export_dir)
```

Defined in `python/saved_model/loader_impl.py`.

Note that the method does not load any data by itself. If the method returns `false`, the export directory definitely does not contain a SavedModel. If the method returns `true`, the export directory may contain a SavedModel but provides no guarantee that it can be loaded.

Args:

- **`export_dir`**: Absolute string path to possible export location. For example, '/my/foo/model'.

Returns:

True if the export directory contains SavedModel files, False otherwise.

tf.saved_model.load

- Contents
- Aliases:
- Used in the guide:

Load a SavedModel from `export_dir`.

Aliases:

- `tf.compat.v1.saved_model.load_v2`
- `tf.compat.v2.saved_model.load`
- `tf.saved_model.load`

```
tf.saved_model.load(
    export_dir,
    tags=None
)
```

Defined in `python/saved_model/load.py`.

Used in the guide:

- Using the SavedModel format

Signatures associated with the SavedModel are available as functions:

```
imported = tf.saved_model.load(path)
f = imported.signatures["serving_default"]
print(f(x=tf.constant([[1.]])))
```

Objects exported with `tf.saved_model.save` additionally have trackable objects and functions assigned to attributes:

```
exported = tf.train.Checkpoint(v=tf.Variable(3.))
exported.f = tf.function(
    lambda x: exported.v * x,
    input_signature=[tf.TensorSpec(shape=None, dtype=tf.float32)])
tf.saved_model.save(exported, path)
imported = tf.saved_model.load(path)
assert 3. == imported.v.numpy()
assert 6. == imported.f(x=tf.constant(2.)).numpy()
```

Loading Keras models

Keras models are trackable, so they can be saved to SavedModel. The object returned by `tf.saved_model.load` is not a Keras object (i.e. doesn't have `.fit`, `.predict`, etc. methods). A few attributes and functions are still available: `.variables`, `.trainable_variables` and `.__call__`.

```
model = tf.keras.Model(...)
tf.saved_model.save(model, path)
```

```
imported = tf.saved_model.load(path)
outputs = imported(inputs)
```

Use `tf.keras.models.load_model` to restore the Keras model.
Importing SavedModels from TensorFlow 1.x
SavedModels from `tf.estimator.Estimator` or 1.x SavedModel APIs have a flat graph instead of `tf.function` objects. These SavedModels will have functions corresponding to their signatures in the `.signatures` attribute, but also have a `.prune` method which allows you to extract functions for new subgraphs. This is equivalent to importing the SavedModel and naming feeds and fetches in a Session from TensorFlow 1.x.

```
imported = tf.saved_model.load(path_to_v1_saved_model)
pruned = imported.prune("x:0", "out:0")
pruned(tf.ones([]))
```

See `tf.compat.v1.wrap_function` for details. These SavedModels also have a `.variables`attribute containing imported variables, and a `.graph` attribute representing the whole imported graph. For SavedModels exported from `tf.saved_model.save`, variables are instead assigned to whichever attributes they were assigned before export.

Args:

- **`export_dir`**: The SavedModel directory to load from.
- **`tags`**: A tag or sequence of tags identifying the MetaGraph to load. Optional if the SavedModel contains a single MetaGraph, as for those exported from `tf.saved_model.load`.

Returns:

A trackable object with a `signatures` attribute mapping from signature keys to functions. If the SavedModel was exported by `tf.saved_model.load`, it also points to trackable objects and functions which were attached to the exported object.

Raises:

- **`ValueError`**: If `tags` don't match a MetaGraph in the SavedModel.

tf.saved_model.save

- Contents
- Aliases:
- Used in the guide:

Exports the Trackable object `obj` to SavedModel format.

Aliases:

- `tf.compat.v1.saved_model.experimental.save`
- `tf.compat.v1.saved_model.save`
- `tf.compat.v2.saved_model.save`
- `tf.saved_model.save`

```
tf.saved_model.save(
    obj,
    export_dir,
    signatures=None
)
```

Defined in `python/saved_model/save.py`.

Used in the guide:

- Using the SavedModel format

Example usage:

```
class Adder(tf.Module):

  @tf.function(input_signature=[tf.TensorSpec(shape=None, dtype=tf.float32)])
  def add(self, x):
    return x + x + 1.

to_export = Adder()
tf.saved_model.save(to_export, '/tmp/adder')
```

The resulting SavedModel is then servable with an input named "x", its value having any shape and dtype float32.

The optional `signatures` argument controls which methods in `obj` will be available to programs which consume `SavedModel`s, for example serving APIs. Python functions may be decorated with `@tf.function(input_signature=...)` and passed as signatures directly, or lazily with a call to `get_concrete_function` on the method decorated with `@tf.function`.

If the `signatures` argument is omitted, `obj` will be searched for `@tf.function`-decorated methods. If exactly one `@tf.function` is found, that method will be used as the default signature for the SavedModel. This behavior is expected to change in the future, when a corresponding `tf.saved_model.load` symbol is added. At that point signatures will be completely optional, and any `@tf.function` attached to `obj` or its dependencies will be exported for use with `load`.

When invoking a signature in an exported SavedModel, `Tensor` arguments are identified by name. These names will come from the Python function's argument names by default. They may be overridden by specifying a `name=...` argument in the corresponding `tf.TensorSpec` object. Explicit naming is required if multiple `Tensor`s are passed through a single argument to the Python function.

The outputs of functions used as `signatures` must either be flat lists, in which case outputs will be numbered, or a dictionary mapping string keys to `Tensor`, in which case the keys will be used to name outputs.

Signatures are available in objects returned by `tf.saved_model.load` as a `.signatures` attribute. This is a reserved attribute: `tf.saved_model.save` on an object with a custom `.signatures` attribute will raise an exception.

Since `tf.keras.Model` objects are also Trackable, this function can be used to export Keras models. For example, exporting with a signature specified:

```
class Model(tf.keras.Model):

  @tf.function(input_signature=[tf.TensorSpec(shape=[None], dtype=tf.string)])
  def serve(self, serialized):
    ...

m = Model()
tf.saved_model.save(m, '/tmp/saved_model/')
```

Exporting from a function without a fixed signature:

```
class Model(tf.keras.Model):

  @tf.function
  def call(self, x):
    ...
```

```
m = Model()
tf.saved_model.save(
    m, '/tmp/saved_model/',
    signatures=m.call.get_concrete_function(
        tf.TensorSpec(shape=[None, 3], dtype=tf.float32, name="inp")))
```

`tf.keras.Model` instances constructed from inputs and outputs already have a signature and so do not require a `@tf.function` decorator or a `signatures` argument. If neither are specified, the model's forward pass is exported.

```
x = input_layer.Input((4,), name="x")
y = core.Dense(5, name="out")(x)
model = training.Model(x, y)
tf.saved_model.save(model, '/tmp/saved_model/')
# The exported SavedModel takes "x" with shape [None, 4] and returns "out"
# with shape [None, 5]
```

Variables must be tracked by assigning them to an attribute of a tracked object or to an attribute of `obj` directly. TensorFlow objects (e.g. layers from `tf.keras.layers`, optimizers from `tf.train`) track their variables automatically. This is the same tracking scheme that `tf.train.Checkpoint`uses, and an exported `Checkpoint` object may be restored as a training checkpoint by pointing `tf.train.Checkpoint.restore` to the SavedModel's "variables/" subdirectory. Currently variables are the only stateful objects supported by `tf.saved_model.save`, but others (e.g. tables) will be supported in the future.

`tf.function` does not hard-code device annotations from outside the function body, instead using the calling context's device. This means for example that exporting a model which runs on a GPU and serving it on a CPU will generally work, with some exceptions. `tf.device` annotations inside the body of the function will be hard-coded in the exported model; this type of annotation is discouraged. Device-specific operations, e.g. with "cuDNN" in the name or with device-specific layouts, may cause issues. Currently a `DistributionStrategy` is another exception: active distribution strategies will cause device placements to be hard-coded in a function. Exporting a single-device computation and importing under a `DistributionStrategy` is not currently supported, but may be in the future.

SavedModels exported with `tf.saved_model.save` strip default-valued attributes automatically, which removes one source of incompatibilities when the consumer of a SavedModel is running an older TensorFlow version than the producer. There are however other sources of incompatibilities which are not handled automatically, such as when the exported model contains operations which the consumer does not have definitions for.

Args:

- **`obj`**: A trackable object to export.
- **`export_dir`**: A directory in which to write the SavedModel.
- **`signatures`**: Optional, either a `tf.function` with an input signature specified or the result of `f.get_concrete_function` on a `@tf.function`-decorated function `f`, in which case `f` will be used to generate a signature for the SavedModel under the default serving signature key. `signatures` may also be a dictionary, in which case it maps from signature keys to either `tf.function` instances with input signatures or concrete functions. The keys of such a dictionary may be arbitrary strings, but will typically be from the`tf.saved_model.signature_constants` module.

Raises:

- **`ValueError`**: If `obj` is not trackable.

Eager Compatibility
Not well supported when graph building. From TensorFlow 1.x,`tf.compat.v1.enable_eager_execution()` should run first. Calling tf.saved_model.save in a loop when graph building from TensorFlow 1.x will add new save operations to the default graph each iteration.
May not be called from within a function body.

tf.compat.v1.saved_model.Builder

- Contents
- Class Builder
  - Aliases:
- __init__
- Methods

Class `Builder`

Builds the `SavedModel` protocol buffer and saves variables and assets.

Aliases:

- Class `tf.compat.v1.saved_model.Builder`
- Class `tf.compat.v1.saved_model.builder.SavedModelBuilder`

Defined in `python/saved_model/builder_impl.py`.
The `SavedModelBuilder` class provides functionality to build a `SavedModel` protocol buffer. Specifically, this allows multiple meta graphs to be saved as part of a single language-neutral `SavedModel`, while sharing variables and assets.
To build a SavedModel, the first meta graph must be saved with variables. Subsequent meta graphs will simply be saved with their graph definitions. If assets need to be saved and written or copied to disk, they can be provided when the meta graph def is added. If multiple meta graph defs are associated an asset of the same name, only the first version is retained.
Each meta graph added to the SavedModel must be annotated with tags. The tags provide a means to identify the specific meta graph to load and restore, along with the shared set of variables and assets.
Typical usage for the `SavedModelBuilder`:

```
...
builder = tf.compat.v1.saved_model.Builder(export_dir)

with tf.compat.v1.Session(graph=tf.Graph()) as sess:
  ...
  builder.add_meta_graph_and_variables(sess,
                                  ["foo-tag"],
                                  signature_def_map=foo_signatures,
                                  assets_collection=foo_assets)
...

with tf.compat.v1.Session(graph=tf.Graph()) as sess:
  ...
  builder.add_meta_graph(["bar-tag", "baz-tag"])
...

builder.save()
```

Note: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.builder.SavedModelBuilder or tf.compat.v1.saved_model.Builder. Tensorflow 2.0 will introduce a new object-based method of creating SavedModels.

`__init__`

```
__init__(export_dir)
```

Methods

`add_meta_graph`

```
add_meta_graph(
    tags,
    signature_def_map=None,
    assets_collection=None,
    legacy_init_op=None,
    clear_devices=False,
    main_op=None,
    strip_default_attrs=False,
    saver=None
)
```

Adds the current meta graph to the SavedModel.
Creates a Saver in the current scope and uses the Saver to export the meta graph def. Invoking this API requires the `add_meta_graph_and_variables()` API to have been invoked before.

Args:

- **`tags`**: The set of tags to annotate the meta graph def with.
- **`signature_def_map`**: The map of signature defs to be added to the meta graph def.
- **`assets_collection`**: Assets to be saved with SavedModel. Note that this list should be a subset of the assets saved as part of the first meta graph in the SavedModel.
- **`clear_devices`**: Set to true if the device info on the default graph should be cleared.
- **`init_op`**: Op or group of ops to execute when the graph is loaded. Note that when the init_op is specified it is run after the restore op at load-time.
- **`train_op`**: Op or group of opts that trains the model when run. This will not be run automatically when the graph is loaded, instead saved in a SignatureDef accessible through the exported MetaGraph.
- **`saver`**: An instance of tf.compat.v1.train.Saver that will be used to export the metagraph. If None, a sharded Saver that restores all variables will be used.

Raises:

- **`AssertionError`**: If the variables for the SavedModel have not been saved yet, or if the graph already contains one or more legacy init ops.

`add_meta_graph_and_variables`

```
add_meta_graph_and_variables(
    sess,
    tags,
    signature_def_map=None,
    assets_collection=None,
    legacy_init_op=None,
    clear_devices=False,
```

```
    main_op=None,
    strip_default_attrs=False,
    saver=None
)
```

Adds the current meta graph to the SavedModel and saves variables.
Creates a Saver to save the variables from the provided session. Exports the corresponding meta graph def. This function assumes that the variables to be saved have been initialized. For a given `SavedModelBuilder`, this API must be called exactly once and for the first meta graph to save. For subsequent meta graph defs to be added, the `add_meta_graph()` API must be used.

Args:

- **`sess`**: The TensorFlow session from which to save the meta graph and variables.
- **`tags`**: The set of tags with which to save the meta graph.
- **`signature_def_map`**: The map of signature def map to add to the meta graph def.
- **`assets_collection`**: Assets to be saved with SavedModel.
- **`clear_devices`**: Set to true if the device info on the default graph should be cleared.
- **`init_op`**: Op or group of ops to execute when the graph is loaded. Note that when the init_op is specified it is run after the restore op at load-time.
- **`train_op`**: Op or group of ops that trains the model when run. This will not be run automatically when the graph is loaded, instead saved in a SignatureDef accessible through the exported MetaGraph.
- **`strip_default_attrs`**: Boolean. If `True`, default-valued attributes will be removed from the NodeDefs. For a detailed guide, see [Stripping Default-Valued Attributes]().
- **`saver`**: An instance of tf.compat.v1.train.Saver that will be used to export the metagraph and save variables. If None, a sharded Saver that restores all variables will be used.

`save`

```
save(as_text=False)
```

Writes a `SavedModel` protocol buffer to disk.
The function writes the SavedModel protocol buffer to the export directory in serialized format.

Args:

- **`as_text`**: Writes the SavedModel protocol buffer in text format to disk. Protocol buffers in text format are useful for debugging, but parsing fails when it encounters an unknown field and so is not forward compatible. This means changes to TensorFlow may prevent deployment of new text format SavedModels to existing serving binaries. Do not deploy `as_text` SavedModels to production.
- Returns:

The path to which the SavedModel protocol buffer was written.

tf.compat.v1.saved_model.build_signature_def

- Contents
- Aliases:

Utility function to build a SignatureDef protocol buffer.

Aliases:

- `tf.compat.v1.saved_model.build_signature_def`
- `tf.compat.v1.saved_model.signature_def_utils.build_signature_def`

```
tf.compat.v1.saved_model.build_signature_def(
    inputs=None,
```

```
    outputs=None,
    method_name=None
)
```

Defined in `python/saved_model/signature_def_utils_impl.py`.

Args:

- **`inputs`**: Inputs of the SignatureDef defined as a proto map of string to tensor info.
- **`outputs`**: Outputs of the SignatureDef defined as a proto map of string to tensor info.
- **`method_name`**: Method name of the SignatureDef as a string.

Returns:

A SignatureDef protocol buffer constructed based on the supplied arguments.

tf.compat.v1.saved_model.build_tensor_info

- Contents
- Aliases:

Utility function to build TensorInfo proto from a Tensor. (deprecated)

Aliases:

- `tf.compat.v1.saved_model.build_tensor_info`
- `tf.compat.v1.saved_model.utils.build_tensor_info`

```
tf.compat.v1.saved_model.build_tensor_info(tensor)
```

Defined in `python/saved_model/utils_impl.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.utils.build_tensor_info or tf.compat.v1.saved_model.build_tensor_info.

Args:

- **`tensor`**: Tensor or SparseTensor whose name, dtype and shape are used to build the TensorInfo. For SparseTensors, the names of the three constituent Tensors are used.

Returns:

A TensorInfo protocol buffer constructed based on the supplied argument.

Raises:

- **`RuntimeError`**: If eager execution is enabled.

tf.compat.v1.saved_model.classification_signature_def

- Contents
- Aliases:

Creates classification signature from given examples and predictions.

Aliases:

- `tf.compat.v1.saved_model.classification_signature_def`
- `tf.compat.v1.saved_model.signature_def_utils.classification_signature_def`

```
tf.compat.v1.saved_model.classification_signature_def(
    examples,
    classes,
    scores
```

```
)
```

Defined in `python/saved_model/signature_def_utils_impl.py`.
This function produces signatures intended for use with the TensorFlow Serving Classify API (tensorflow_serving/apis/prediction_service.proto), and so constrains the input and output types to those allowed by TensorFlow Serving.

Args:

- **`examples`**: A string `Tensor`, expected to accept serialized tf.Examples.
- **`classes`**: A string `Tensor`. Note that the ClassificationResponse message requires that class labels are strings, not integers or anything else.
- **`scores`**: a float `Tensor`.

Returns:
A classification-flavored signature_def.

Raises:

- **`ValueError`**: If examples is `None`.

tf.compat.v1.saved_model.contains_saved_model

- Contents
- Aliases:

Checks whether the provided export directory could contain a SavedModel.

Aliases:

- `tf.compat.v1.saved_model.contains_saved_model`
- `tf.compat.v1.saved_model.loader.maybe_saved_model_directory`
- `tf.compat.v1.saved_model.maybe_saved_model_directory`

```
tf.compat.v1.saved_model.contains_saved_model(export_dir)
```

Defined in `python/saved_model/loader_impl.py`.
Note that the method does not load any data by itself. If the method returns `false`, the export directory definitely does not contain a SavedModel. If the method returns `true`, the export directory may contain a SavedModel but provides no guarantee that it can be loaded.

Args:

- **`export_dir`**: Absolute string path to possible export location. For example, '/my/foo/model'.

Returns:
True if the export directory contains SavedModel files, False otherwise.

tf.compat.v1.saved_model.get_tensor_from_tensor_info

- Contents
- Aliases:

Returns the Tensor or SparseTensor described by a TensorInfo proto. (deprecated)

Aliases:

- `tf.compat.v1.saved_model.get_tensor_from_tensor_info`
- `tf.compat.v1.saved_model.utils.get_tensor_from_tensor_info`

```
tf.compat.v1.saved_model.get_tensor_from_tensor_info(
    tensor_info,
    graph=None,
    import_scope=None
)
```

Defined in `python/saved_model/utils_impl.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.utils.get_tensor_from_tensor_info or tf.compat.v1.saved_model.get_tensor_from_tensor_info.

Args:

- **`tensor_info`**: A TensorInfo proto describing a Tensor or SparseTensor.
- **`graph`**: The tf.Graph in which tensors are looked up. If None, the current default graph is used.
- **`import_scope`**: If not None, names in `tensor_info` are prefixed with this string before lookup.

Returns:

The Tensor or SparseTensor in `graph` described by `tensor_info`.

Raises:

- **`KeyError`**: If `tensor_info` does not correspond to a tensor in `graph`.
- **`ValueError`**: If `tensor_info` is malformed.

tf.compat.v1.saved_model.is_valid_signature

- Contents
- Aliases:

Determine whether a SignatureDef can be served by TensorFlow Serving.

Aliases:

- `tf.compat.v1.saved_model.is_valid_signature`
- `tf.compat.v1.saved_model.signature_def_utils.is_valid_signature`

```
tf.compat.v1.saved_model.is_valid_signature(signature_def)
```

Defined in `python/saved_model/signature_def_utils_impl.py`.

tf.compat.v1.saved_model.load

- Contents
- Aliases:

Loads the model from a SavedModel as specified by tags. (deprecated)

Aliases:

- `tf.compat.v1.saved_model.load`
- `tf.compat.v1.saved_model.loader.load`

```
tf.compat.v1.saved_model.load(
    sess,
    tags,
    export_dir,
    import_scope=None,
    **saver_kwargs
)
```

Defined in `python/saved_model/loader_impl.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.loader.load or tf.compat.v1.saved_model.load. There will be a new function for importing SavedModels in Tensorflow 2.0.

Args:

- **`sess`**: The TensorFlow session to restore the variables.
- **`tags`**: Set of string tags to identify the required MetaGraphDef. These should correspond to the tags used when saving the variables using the SavedModel `save()` API.
- **`export_dir`**: Directory in which the SavedModel protocol buffer and variables to be loaded are located.
- **`import_scope`**: Optional `string` -- if specified, prepend this string followed by '/' to all loaded tensor names. This scope is applied to tensor instances loaded into the passed session, but it is *not* written through to the static `MetaGraphDef` protocol buffer that is returned.
- **`**saver_kwargs`**: Optional keyword arguments passed through to Saver.

Returns:

The `MetaGraphDef` protocol buffer loaded in the provided session. This can be used to further extract signature-defs, collection-defs, etc.

Raises:

- **`RuntimeError`**: MetaGraphDef associated with the tags cannot be found.

tf.compat.v1.saved_model.main_op_with_restore

- Contents
- Aliases:

Returns a main op to init variables, tables and restore the graph. (deprecated)

Aliases:

- `tf.compat.v1.saved_model.main_op.main_op_with_restore`
- `tf.compat.v1.saved_model.main_op_with_restore`

```
tf.compat.v1.saved_model.main_op_with_restore(restore_op_name)
```

Defined in `python/saved_model/main_op_impl.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.main_op_with_restore or tf.compat.v1.saved_model.main_op.main_op_with_restore.

Returns the main op including the group of ops that initializes all variables, initialize local variables, initialize all tables and the restore op name.

Args:

- **`restore_op_name`**: Name of the op to use to restore the graph.

Returns:

The set of ops to be run as part of the main op upon the load operation.

tf.compat.v1.saved_model.predict_signature_def

- Contents
- Aliases:

Creates prediction signature from given inputs and outputs.

Aliases:

- `tf.compat.v1.saved_model.predict_signature_def`
- `tf.compat.v1.saved_model.signature_def_utils.predict_signature_def`

```
tf.compat.v1.saved_model.predict_signature_def(
    inputs,
    outputs
)
```

Defined in `python/saved_model/signature_def_utils_impl.py`.
This function produces signatures intended for use with the TensorFlow Serving Predict API (tensorflow_serving/apis/prediction_service.proto). This API imposes no constraints on the input and output types.

Args:

- **`inputs`**: dict of string to `Tensor`.
- **`outputs`**: dict of string to `Tensor`.

Returns:
A prediction-flavored signature_def.

Raises:

- **`ValueError`**: If inputs or outputs is `None`.

tf.compat.v1.saved_model.regression_signature_def

- Contents
- Aliases:

Creates regression signature from given examples and predictions.

Aliases:

- `tf.compat.v1.saved_model.regression_signature_def`
- `tf.compat.v1.saved_model.signature_def_utils.regression_signature_def`

```
tf.compat.v1.saved_model.regression_signature_def(
    examples,
    predictions
)
```

Defined in `python/saved_model/signature_def_utils_impl.py`.
This function produces signatures intended for use with the TensorFlow Serving Regress API (tensorflow_serving/apis/prediction_service.proto), and so constrains the input and output types to those allowed by TensorFlow Serving.

Args:

- **`examples`**: A string `Tensor`, expected to accept serialized tf.Examples.
- **`predictions`**: A float `Tensor`.

Returns:
A regression-flavored signature_def.

Raises:

- **`ValueError`**: If examples is `None`.

tf.compat.v1.saved_model.simple_save

Convenience function to build a SavedModel suitable for serving. (deprecated)

```
tf.compat.v1.saved_model.simple_save(
    session,
    export_dir,
    inputs,
    outputs,
    legacy_init_op=None
)
```

Defined in `python/saved_model/simple_save.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.simple_save.

In many common cases, saving models for serving will be as simple as:

```
simple_save(session,
            export_dir,
            inputs={"x": x, "y": y},
            outputs={"z": z})
```

Although in many cases it's not necessary to understand all of the many ways to configure a SavedModel, this method has a few practical implications: - It will be treated as a graph for inference / serving (i.e. uses the tag `saved_model.SERVING`) - The SavedModel will load in TensorFlow Serving and supports the Predict API. To use the Classify, Regress, or MultiInference APIs, please use eithertf.Estimator or the lower level SavedModel APIs. - Some TensorFlow ops depend on information on disk or other information called "assets". These are generally handled automatically by adding the assets to the `GraphKeys.ASSET_FILEPATHS` collection. Only assets in that collection are exported; if you need more custom behavior, you'll need to use the SavedModelBuilder.

More information about SavedModel and signatures can be found here: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/saved_model/README.md.

Args:

- **`session`**: The TensorFlow session from which to save the meta graph and variables.
- **`export_dir`**: The path to which the SavedModel will be stored.
- **`inputs`**: dict mapping string input names to tensors. These are added to the SignatureDef as the inputs.
- **`outputs`**: dict mapping string output names to tensors. These are added to the SignatureDef as the outputs.
- **`legacy_init_op`**: Legacy support for op or group of ops to execute after the restore op upon a load.

tf.compat.v1.saved_model.main_op.main_op

Returns a main op to init variables and tables. (deprecated)

```
tf.compat.v1.saved_model.main_op.main_op()
```

Defined in `python/saved_model/main_op_impl.py`.

Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: This function will only be available through the v1 compatibility library as tf.compat.v1.saved_model.main_op.main_op.

Returns the main op including the group of ops that initializes all variables, initializes local variables and initialize all tables.

Returns:

The set of ops to be run as part of the main op upon the load operation.

Module: tf.sets

- **Contents**
- Functions

Tensorflow set operations.

Functions

`difference(...)`: Compute set difference of elements in last dimension of `a` and `b`.

`intersection(...)`: Compute set intersection of elements in last dimension of `a` and `b`.

`size(...)`: Compute number of unique elements along last dimension of `a`.

`union(...)`: Compute set union of elements in last dimension of `a` and `b`.

tf.sets.difference

- Contents
- Aliases:

Compute set difference of elements in last dimension of `a` and `b`.

Aliases:

- `tf.compat.v1.sets.difference`
- `tf.compat.v1.sets.set_difference`
- `tf.compat.v2.sets.difference`
- `tf.sets.difference`

```
tf.sets.difference(
    a,
    b,
    aminusb=True,
    validate_indices=True
)
```

Defined in `python/ops/sets_impl.py`.

All but the last dimension of `a` and `b` must match.

Example:

```
  import tensorflow as tf
  import collections

  # Represent the following array of sets as a sparse tensor:
  # a = np.array([[{1, 2}, {3}], [{4}, {5, 6}]])
  a = collections.OrderedDict([
      ((0, 0, 0), 1),
      ((0, 0, 1), 2),
      ((0, 1, 0), 3),
      ((1, 0, 0), 4),
      ((1, 1, 0), 5),
      ((1, 1, 1), 6),
  ])
```

```
a = tf.SparseTensor(list(a.keys()), list(a.values()), dense_shape=[2, 2, 2])

# np.array([[{1, 3}, {2}], [{4, 5}, {5, 6, 7, 8}]])
b = collections.OrderedDict([
    ((0, 0, 0), 1),
    ((0, 0, 1), 3),
    ((0, 1, 0), 2),
    ((1, 0, 0), 4),
    ((1, 0, 1), 5),
    ((1, 1, 0), 5),
    ((1, 1, 1), 6),
    ((1, 1, 2), 7),
    ((1, 1, 3), 8),
])
b = tf.SparseTensor(list(b.keys()), list(b.values()), dense_shape=[2, 2, 4])

# `set_difference` is applied to each aligned pair of sets.
tf.sets.difference(a, b)

# The result will be equivalent to either of:
#
# np.array([[{2}, {3}], [{}, {}]])
#
# collections.OrderedDict([
#     ((0, 0, 0), 2),
#     ((0, 1, 0), 3),
# ])
```

Args:

- **a**: `Tensor` or `SparseTensor` of the same type as `b`. If sparse, indices must be sorted in row-major order.
- **b**: `Tensor` or `SparseTensor` of the same type as `a`. If sparse, indices must be sorted in row-major order.
- **aminusb**: Whether to subtract `b` from `a`, vs vice versa.
- **validate_indices**: Whether to validate the order and range of sparse indices in `a` and `b`.

Returns:

A `SparseTensor` whose shape is the same rank as `a` and `b`, and all but the last dimension the same. Elements along the last dimension contain the differences.

tf.sets.intersection

- Contents
- Aliases:

Compute set intersection of elements in last dimension of `a` and `b`.

Aliases:

- `tf.compat.v1.sets.intersection`
- `tf.compat.v1.sets.set_intersection`
- `tf.compat.v2.sets.intersection`
- `tf.sets.intersection`

```
tf.sets.intersection(
    a,
    b,
    validate_indices=True
)
```

Defined in `python/ops/sets_impl.py`.
All but the last dimension of `a` and `b` must match.

Example:

```
  import tensorflow as tf
  import collections

  # Represent the following array of sets as a sparse tensor:
  # a = np.array([[{1, 2}, {3}], [{4}, {5, 6}]])
  a = collections.OrderedDict([
      ((0, 0, 0), 1),
      ((0, 0, 1), 2),
      ((0, 1, 0), 3),
      ((1, 0, 0), 4),
      ((1, 1, 0), 5),
      ((1, 1, 1), 6),
  ])
  a = tf.SparseTensor(list(a.keys()), list(a.values()), dense_shape=[2,2,2])

  # b = np.array([[{1}, {}], [{4}, {5, 6, 7, 8}]])
  b = collections.OrderedDict([
      ((0, 0, 0), 1),
      ((1, 0, 0), 4),
      ((1, 1, 0), 5),
      ((1, 1, 1), 6),
      ((1, 1, 2), 7),
      ((1, 1, 3), 8),
  ])
  b = tf.SparseTensor(list(b.keys()), list(b.values()), dense_shape=[2, 2, 4])

  # `tf.sets.intersection` is applied to each aligned pair of sets.
  tf.sets.intersection(a, b)

  # The result will be equivalent to either of:
  #
  # np.array([[{1}, {}], [{4}, {5, 6}]])
  #
  # collections.OrderedDict([
  #     ((0, 0, 0), 1),
  #     ((1, 0, 0), 4),
  #     ((1, 1, 0), 5),
  #     ((1, 1, 1), 6),
```

```
  # ])
```

Args:

- **a**: `Tensor` or `SparseTensor` of the same type as `b`. If sparse, indices must be sorted in row-major order.
- **b**: `Tensor` or `SparseTensor` of the same type as `a`. If sparse, indices must be sorted in row-major order.
- **validate_indices**: Whether to validate the order and range of sparse indices in `a` and `b`.

Returns:

A `SparseTensor` whose shape is the same rank as `a` and `b`, and all but the last dimension the same. Elements along the last dimension contain the intersections.

tf.sets.size

- Contents
- Aliases:

Compute number of unique elements along last dimension of `a`.

Aliases:

- `tf.compat.v1.sets.set_size`
- `tf.compat.v1.sets.size`
- `tf.compat.v2.sets.size`
- `tf.sets.size`

```
tf.sets.size(
    a,
    validate_indices=True
)
```

Defined in `python/ops/sets_impl.py`.

Args:

- **a**: `SparseTensor`, with indices sorted in row-major order.
- **validate_indices**: Whether to validate the order and range of sparse indices in `a`.

Returns:

`int32` `Tensor` of set sizes. For `a` ranked `n`, this is a `Tensor` with rank `n-1`, and the same 1st `n-1`dimensions as `a`. Each value is the number of unique elements in the corresponding `[0...n-1]`dimension of `a`.

Raises:

- **TypeError**: If `a` is an invalid types.

tf.sets.union

- Contents
- Aliases:

Compute set union of elements in last dimension of `a` and `b`.

Aliases:

- `tf.compat.v1.sets.set_union`
- `tf.compat.v1.sets.union`
- `tf.compat.v2.sets.union`
- `tf.sets.union`

```
tf.sets.union(
    a,
    b,
    validate_indices=True
)
```

Defined in `python/ops/sets_impl.py`.
All but the last dimension of `a` and `b` must match.

Example:

```
import tensorflow as tf
import collections

# [[{1, 2}, {3}], [{4}, {5, 6}]]
a = collections.OrderedDict([
    ((0, 0, 0), 1),
    ((0, 0, 1), 2),
    ((0, 1, 0), 3),
    ((1, 0, 0), 4),
    ((1, 1, 0), 5),
    ((1, 1, 1), 6),
])
a = tf.SparseTensor(list(a.keys()), list(a.values()), dense_shape=[2, 2, 2])

# [[{1, 3}, {2}], [{4, 5}, {5, 6, 7, 8}]]
b = collections.OrderedDict([
    ((0, 0, 0), 1),
    ((0, 0, 1), 3),
    ((0, 1, 0), 2),
    ((1, 0, 0), 4),
    ((1, 0, 1), 5),
    ((1, 1, 0), 5),
    ((1, 1, 1), 6),
    ((1, 1, 2), 7),
    ((1, 1, 3), 8),
])
b = tf.SparseTensor(list(b.keys()), list(b.values()), dense_shape=[2, 2, 4])

# `set_union` is applied to each aligned pair of sets.
tf.sets.union(a, b)

# The result will be a equivalent to either of:
#
# np.array([[{1, 2, 3}, {2, 3}], [{4, 5}, {5, 6, 7, 8}]])
#
# collections.OrderedDict([
#     ((0, 0, 0), 1),
#     ((0, 0, 1), 2),
#     ((0, 0, 2), 3),
```

```
#     ((0, 1, 0), 2),
#     ((0, 1, 1), 3),
#     ((1, 0, 0), 4),
#     ((1, 0, 1), 5),
#     ((1, 1, 0), 5),
#     ((1, 1, 1), 6),
#     ((1, 1, 2), 7),
#     ((1, 1, 3), 8),
# ])
```

Args:

- **a**: `Tensor` or `SparseTensor` of the same type as `b`. If sparse, indices must be sorted in row-major order.
- **b**: `Tensor` or `SparseTensor` of the same type as `a`. If sparse, indices must be sorted in row-major order.
- **validate_indices**: Whether to validate the order and range of sparse indices in `a` and `b`.

Returns:
A `SparseTensor` whose shape is the same rank as `a` and `b`, and all but the last dimension the same. Elements along the last dimension contain the unions.

Module: tf.signal

- **Contents**
- Functions

Signal processing operations.
See the tf.signal guide.

Functions

`dct(...)`: Computes the 1D [Discrete Cosine Transform (DCT)][dct] of `input`.
`fft(...)`: Fast Fourier transform.
`fft2d(...)`: 2D fast Fourier transform.
`fft3d(...)`: 3D fast Fourier transform.
`fftshift(...)`: Shift the zero-frequency component to the center of the spectrum.
`frame(...)`: Expands `signal`'s `axis` dimension into frames of `frame_length`.
`hamming_window(...)`: Generate a Hamming window.
`hann_window(...)`: Generate a Hann window.
`idct(...)`: Computes the 1D [Inverse Discrete Cosine Transform (DCT)][idct] of `input`.
`ifft(...)`: Inverse fast Fourier transform.
`ifft2d(...)`: Inverse 2D fast Fourier transform.
`ifft3d(...)`: Inverse 3D fast Fourier transform.
`ifftshift(...)`: The inverse of fftshift.
`inverse_stft(...)`: Computes the inverse Short-time Fourier Transform of `stfts`.
`inverse_stft_window_fn(...)`: Generates a window function that can be used in `inverse_stft`.
`irfft(...)`: Inverse real-valued fast Fourier transform.
`irfft2d(...)`: Inverse 2D real-valued fast Fourier transform.
`irfft3d(...)`: Inverse 3D real-valued fast Fourier transform.
`linear_to_mel_weight_matrix(...)`: Returns a matrix to warp linear scale spectrograms to the mel scale.
`mfccs_from_log_mel_spectrograms(...)`: Computes MFCCs of `log_mel_spectrograms`.
`overlap_and_add(...)`: Reconstructs a signal from a framed representation.
`rfft(...)`: Real-valued fast Fourier transform.

`rfft2d(...)`: 2D real-valued fast Fourier transform.
`rfft3d(...)`: 3D real-valued fast Fourier transform.
`stft(...)`: Computes the Short-time Fourier Transform of `signals`.

tf.signal.dct

- Contents
- Aliases:

Computes the 1D Discrete Cosine Transform (DCT) of `input`.

Aliases:

- `tf.compat.v1.signal.dct`
- `tf.compat.v1.spectral.dct`
- `tf.compat.v2.signal.dct`
- `tf.signal.dct`

```
tf.signal.dct(
    input,
    type=2,
    n=None,
    axis=-1,
    norm=None,
    name=None
)
```

Defined in `python/ops/signal/dct_ops.py`.
Currently only Types I, II and III are supported. Type I is implemented using a length `2N` padded `tf.signal.rfft`. Type II is implemented using a length `2N` padded `tf.signal.rfft`, as described here: Type 2 DCT using 2N FFT padded (Makhoul). Type III is a fairly straightforward inverse of Type II (i.e. using a length `2N` padded `tf.signal.irfft`).

Args:

- **`input`**: A `[..., samples]` `float32` `Tensor` containing the signals to take the DCT of.
- **`type`**: The DCT type to perform. Must be 1, 2 or 3.
- **`n`**: The length of the transform. If length is less than sequence length, only the first n elements of the sequence are considered for the DCT. If n is greater than the sequence length, zeros are padded and then the DCT is computed as usual.
- **`axis`**: For future expansion. The axis to compute the DCT along. Must be `-1`.
- **`norm`**: The normalization to apply. `None` for no normalization or `'ortho'` for orthonormal normalization.
- **`name`**: An optional name for the operation.

Returns:

A `[..., samples]` `float32` `Tensor` containing the DCT of `input`.

Raises:

- **`ValueError`**: If `type` is not `1`, `2` or `3`, `axis` is not `-1`, `n` is not `None` or greater than 0, or `norm` is not `None` or `'ortho'`.
- **`ValueError`**: If `type` is `1` and `norm` is `ortho`.

Scipy Compatibility

Equivalent to scipy.fftpack.dct for Type-I, Type-II and Type-III DCT.

tf.signal.fft

- Contents
- Aliases:

Fast Fourier transform.

Aliases:
- `tf.compat.v1.fft`
- `tf.compat.v1.signal.fft`
- `tf.compat.v1.spectral.fft`
- `tf.compat.v2.signal.fft`
- `tf.signal.fft`

```
tf.signal.fft(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.

Computes the 1-dimensional discrete Fourier transform over the inner-most dimension of `input`.

Args:
- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex tensor.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.signal.fft2d

- Contents
- Aliases:

2D fast Fourier transform.

Aliases:
- `tf.compat.v1.fft2d`
- `tf.compat.v1.signal.fft2d`
- `tf.compat.v1.spectral.fft2d`
- `tf.compat.v2.signal.fft2d`
- `tf.signal.fft2d`

```
tf.signal.fft2d(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.

Computes the 2-dimensional discrete Fourier transform over the inner-most 2 dimensions of `input`.

Args:
- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex tensor.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.signal.fft3d

- Contents

- Aliases:

3D fast Fourier transform.

Aliases:

- `tf.compat.v1.fft3d`
- `tf.compat.v1.signal.fft3d`
- `tf.compat.v1.spectral.fft3d`
- `tf.compat.v2.signal.fft3d`
- `tf.signal.fft3d`

```
tf.signal.fft3d(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.
Computes the 3-dimensional discrete Fourier transform over the inner-most 3 dimensions of `input`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex64 tensor.
- **`name`**: A name for the operation (optional).

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

tf.signal.fftshift

- Contents
- Aliases:

Shift the zero-frequency component to the center of the spectrum.

Aliases:

- `tf.compat.v1.signal.fftshift`
- `tf.compat.v2.signal.fftshift`
- `tf.signal.fftshift`

```
tf.signal.fftshift(
    x,
    axes=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.
This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

For example:

```
x = tf.signal.fftshift([ 0.,  1.,  2.,  3.,  4., -5., -4., -3., -2., -1.])
x.numpy() # array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Args:

- **`x`**: `Tensor`, input tensor.
- **`axes`**: `int` or shape `tuple`, optional Axes over which to shift. Default is None, which shifts all axes.
- **`name`**: An optional name for the operation.

Returns:
A `Tensor`, The shifted tensor.

Numpy Compatibility
Equivalent to numpy.fft.fftshift.
https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fftshift.html

tf.signal.frame

- Contents
- Aliases:

Expands `signal`'s `axis` dimension into frames of `frame_length`.

Aliases:

- `tf.compat.v1.signal.frame`
- `tf.compat.v2.signal.frame`
- `tf.signal.frame`

```
tf.signal.frame(
    signal,
    frame_length,
    frame_step,
    pad_end=False,
    pad_value=0,
    axis=-1,
    name=None
)
```

Defined in `python/ops/signal/shape_ops.py`.
Slides a window of size `frame_length` over `signal`'s `axis` dimension with a stride of `frame_step`, replacing the `axis` dimension with `[frames, frame_length]` frames.
If `pad_end` is True, window positions that are past the end of the `axis` dimension are padded with `pad_value` until the window moves fully past the end of the dimension. Otherwise, only window positions that fully overlap the `axis` dimension are produced.

For example:

```
pcm = tf.compat.v1.placeholder(tf.float32, [None, 9152])
frames = tf.signal.frame(pcm, 512, 180)
magspec = tf.abs(tf.signal.rfft(frames, [512]))
image = tf.expand_dims(magspec, 3)
```

Args:

- **`signal`**: A `[..., samples, ...]` `Tensor`. The rank and dimensions may be unknown. Rank must be at least 1.
- **`frame_length`**: The frame length in samples. An integer or scalar `Tensor`.
- **`frame_step`**: The frame hop size in samples. An integer or scalar `Tensor`.
- **`pad_end`**: Whether to pad the end of `signal` with `pad_value`.
- **`pad_value`**: An optional scalar `Tensor` to use where the input signal does not exist when `pad_end` is True.
- **`axis`**: A scalar integer `Tensor` indicating the axis to frame. Defaults to the last axis. Supports negative values for indexing from the end.
- **`name`**: An optional name for the operation.

Returns:
A `Tensor` of frames with shape `[..., frames, frame_length, ...]`.

Raises:

- **`ValueError`**: If `frame_length`, `frame_step`, `pad_value`, or `axis` are not scalar.

tf.signal.hamming_window

- Contents
- Aliases:

Generate a [Hamming](https://en.wikipedia.org/wiki/Window_function#Hann_and_Hamming_windows) window.

Aliases:

- `tf.compat.v1.signal.hamming_window`
- `tf.compat.v2.signal.hamming_window`
- `tf.signal.hamming_window`

```
tf.signal.hamming_window(
    window_length,
    periodic=True,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in `python/ops/signal/window_ops.py`.

Args:

- **`window_length`**: A scalar `Tensor` indicating the window length to generate.
- **`periodic`**: A bool `Tensor` indicating whether to generate a periodic or symmetric window. Periodic windows are typically used for spectral analysis while symmetric windows are typically used for digital filter design.
- **`dtype`**: The data type to produce. Must be a floating point type.
- **`name`**: An optional name for the operation.

Returns:
A `Tensor` of shape `[window_length]` of type `dtype`.

Raises:

- **`ValueError`**: If `dtype` is not a floating point type.

tf.signal.hann_window

- Contents
- Aliases:

Generate a Hann window.

Aliases:

- `tf.compat.v1.signal.hann_window`
- `tf.compat.v2.signal.hann_window`
- `tf.signal.hann_window`

```
tf.signal.hann_window(
    window_length,
    periodic=True,
    dtype=tf.dtypes.float32,
    name=None
```

```
)
```

Defined in `python/ops/signal/window_ops.py`.

Args:

- **`window_length`**: A scalar `Tensor` indicating the window length to generate.
- **`periodic`**: A bool `Tensor` indicating whether to generate a periodic or symmetric window. Periodic windows are typically used for spectral analysis while symmetric windows are typically used for digital filter design.
- **`dtype`**: The data type to produce. Must be a floating point type.
- **`name`**: An optional name for the operation.

Returns:

A `Tensor` of shape `[window_length]` of type `dtype`.

Raises:

- **`ValueError`**: If `dtype` is not a floating point type.

tf.signal.idct

- Contents
- Aliases:

Computes the 1D Inverse Discrete Cosine Transform (DCT) of `input`.

Aliases:

- `tf.compat.v1.signal.idct`
- `tf.compat.v1.spectral.idct`
- `tf.compat.v2.signal.idct`
- `tf.signal.idct`

```
tf.signal.idct(
    input,
    type=2,
    n=None,
    axis=-1,
    norm=None,
    name=None
)
```

Defined in `python/ops/signal/dct_ops.py`.

Currently only Types I, II and III are supported. Type III is the inverse of Type II, and vice versa.

Note that you must re-normalize by 1/(2n) to obtain an inverse if `norm` is not `'ortho'`. That is:`signal == idct(dct(signal)) * 0.5 / signal.shape[-1]`. When `norm='ortho'`, we have:`signal == idct(dct(signal, norm='ortho'), norm='ortho')`.

Args:

- **`input`**: A `[..., samples]` `float32` `Tensor` containing the signals to take the DCT of.
- **`type`**: The IDCT type to perform. Must be 1, 2 or 3.
- **`n`**: For future expansion. The length of the transform. Must be `None`.
- **`axis`**: For future expansion. The axis to compute the DCT along. Must be `-1`.
- **`norm`**: The normalization to apply. `None` for no normalization or `'ortho'` for orthonormal normalization.
- **`name`**: An optional name for the operation.

Returns:

A `[..., samples]` `float32` `Tensor` containing the IDCT of `input`.

Raises:

- **`ValueError`**: If `type` is not `1`, `2` or `3`, `n` is not `None`,`axis`is not`-1`, or`norm`is not`None`or`'ortho'`.

Scipy Compatibility

Equivalent to scipy.fftpack.idct for Type-I, Type-II and Type-III DCT.

tf.signal.ifft

- Contents
- Aliases:

Inverse fast Fourier transform.

Aliases:

- `tf.compat.v1.ifft`
- `tf.compat.v1.signal.ifft`
- `tf.compat.v1.spectral.ifft`
- `tf.compat.v2.signal.ifft`
- `tf.signal.ifft`

```
tf.signal.ifft(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.

Computes the inverse 1-dimensional discrete Fourier transform over the inner-most dimension of `input`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex tensor.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.signal.ifft2d

- Contents
- Aliases:

Inverse 2D fast Fourier transform.

Aliases:

- `tf.compat.v1.ifft2d`
- `tf.compat.v1.signal.ifft2d`
- `tf.compat.v1.spectral.ifft2d`
- `tf.compat.v2.signal.ifft2d`
- `tf.signal.ifft2d`

```
tf.signal.ifft2d(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.

Computes the inverse 2-dimensional discrete Fourier transform over the inner-most 2 dimensions of `input`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex tensor.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.signal.ifft3d

- Contents
- Aliases:

Inverse 3D fast Fourier transform.

Aliases:

- `tf.compat.v1.ifft3d`
- `tf.compat.v1.signal.ifft3d`
- `tf.compat.v1.spectral.ifft3d`
- `tf.compat.v2.signal.ifft3d`
- `tf.signal.ifft3d`

```
tf.signal.ifft3d(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_spectral_ops.py`.

Computes the inverse 3-dimensional discrete Fourier transform over the inner-most 3 dimensions of `input`.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `complex64`, `complex128`. A complex64 tensor.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.signal.ifftshift

- Contents
- Aliases:

The inverse of fftshift.

Aliases:

- `tf.compat.v1.signal.ifftshift`
- `tf.compat.v2.signal.ifftshift`
- `tf.signal.ifftshift`

```
tf.signal.ifftshift(
    x,
    axes=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.

Although identical for even-length x, the functions differ by one sample for odd-length x.

For example:

```
x = tf.signal.ifftshift([[ 0.,  1.,  2.],[ 3.,  4., -4.],[-3., -2., -1.]])
x.numpy() # array([[ 4., -4.,  3.],[-2., -1., -3.],[ 1.,  2.,  0.]])
```

Args:

- **x**: `Tensor`, input tensor.
- **axes**: `int` or shape `tuple` Axes over which to calculate. Defaults to None, which shifts all axes.
- **name**: An optional name for the operation.

Returns:

A `Tensor`, The shifted tensor.

Numpy Compatibility

Equivalent to numpy.fft.ifftshift.
https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.ifftshift.html

tf.signal.inverse_stft

- Contents
- Aliases:

Computes the inverse Short-time Fourier Transform of `stfts`.

Aliases:

- `tf.compat.v1.signal.inverse_stft`
- `tf.compat.v2.signal.inverse_stft`
- `tf.signal.inverse_stft`

```
tf.signal.inverse_stft(
    stfts,
    frame_length,
    frame_step,
    fft_length=None,
    window_fn=tf.signal.hann_window,
    name=None
)
```

Defined in `python/ops/signal/spectral_ops.py`.

To reconstruct an original waveform, a complimentary window function should be used in inverse_stft. Such a window function can be constructed with tf.signal.inverse_stft_window_fn.

Example:

```
frame_length = 400
frame_step = 160
waveform = tf.compat.v1.placeholder(dtype=tf.float32, shape=[1000])
stft = tf.signal.stft(waveform, frame_length, frame_step)
inverse_stft = tf.signal.inverse_stft(
    stft, frame_length, frame_step,
    window_fn=tf.signal.inverse_stft_window_fn(frame_step))
```

if a custom window_fn is used in stft, it must be passed to inverse_stft_window_fn:

```
frame_length = 400
frame_step = 160
window_fn = functools.partial(window_ops.hamming_window, periodic=True),
```

```
waveform = tf.compat.v1.placeholder(dtype=tf.float32, shape=[1000])
stft = tf.signal.stft(
    waveform, frame_length, frame_step, window_fn=window_fn)
inverse_stft = tf.signal.inverse_stft(
    stft, frame_length, frame_step,
    window_fn=tf.signal.inverse_stft_window_fn(
       frame_step, forward_window_fn=window_fn))
```

Implemented with GPU-compatible ops and supports gradients.

Args:

- **`stfts`**: A `complex64` `[..., frames, fft_unique_bins]` `Tensor` of STFT bins representing a batch of `fft_length`-point STFTs where `fft_unique_bins` is `fft_length // 2 + 1`
- **`frame_length`**: An integer scalar `Tensor`. The window length in samples.
- **`frame_step`**: An integer scalar `Tensor`. The number of samples to step.
- **`fft_length`**: An integer scalar `Tensor`. The size of the FFT that produced `stfts`. If not provided, uses the smallest power of 2 enclosing `frame_length`.
- **`window_fn`**: A callable that takes a window length and a `dtype` keyword argument and returns a `[window_length]` `Tensor` of samples in the provided datatype. If set to `None`, no windowing is used.
- **`name`**: An optional name for the operation.

Returns:

A `[..., samples]` `Tensor` of `float32` signals representing the inverse STFT for each input STFT in `stfts`.

Raises:

- **`ValueError`**: If `stfts` is not at least rank 2, `frame_length` is not scalar, `frame_step` is not scalar, or `fft_length` is not scalar.

tf.signal.inverse_stft_window_fn

- Contents
- Aliases:

Generates a window function that can be used in `inverse_stft`.

Aliases:

- `tf.compat.v1.signal.inverse_stft_window_fn`
- `tf.compat.v2.signal.inverse_stft_window_fn`
- `tf.signal.inverse_stft_window_fn`

```
tf.signal.inverse_stft_window_fn(
    frame_step,
    forward_window_fn=tf.signal.hann_window,
    name=None
)
```

Defined in `python/ops/signal/spectral_ops.py`.

Constructs a window that is equal to the forward window with a further pointwise amplitude correction. `inverse_stft_window_fn` is equivalent to `forward_window_fn` in the case where it would produce an exact inverse.

See examples in `inverse_stft` documentation for usage.

Args:

- **`frame_step`**: An integer scalar `Tensor`. The number of samples to step.

- **forward_window_fn**: window_fn used in the forward transform, `stft`.
- **name**: An optional name for the operation.

Returns:
A callable that takes a window length and a `dtype` keyword argument and returns a `[window_length] Tensor` of samples in the provided datatype. The returned window is suitable for reconstructing original waveform in inverse_stft.

tf.signal.irfft

- Contents
- Aliases:

Inverse real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.irfft`
- `tf.compat.v1.spectral.irfft`
- `tf.compat.v2.signal.irfft`
- `tf.signal.irfft`

```
tf.signal.irfft(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.
Computes the inverse 1-dimensional discrete Fourier transform of a real-valued signal over the inner-most dimension of `input`.
The inner-most dimension of `input` is assumed to be the result of `RFFT`: the `fft_length / 2 + 1`unique components of the DFT of a real-valued signal. If `fft_length` is not provided, it is computed from the size of the inner-most dimension of `input` (`fft_length = 2 * (inner - 1)`). If the FFT length used to compute `input` is odd, it should be provided since it cannot be inferred properly.
Along the axis `IRFFT` is computed on, if `fft_length / 2 + 1` is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **input**: A `Tensor` of type `complex64`. A complex64 tensor.
- **fft_length**: A `Tensor` of type `int32`. An int32 tensor of shape [1]. The FFT length.
- **name**: A name for the operation (optional).

Returns:
A `Tensor` of type `float32`.

tf.signal.irfft2d

- Contents
- Aliases:

Inverse 2D real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.irfft2d`
- `tf.compat.v1.spectral.irfft2d`
- `tf.compat.v2.signal.irfft2d`
- `tf.signal.irfft2d`

```
tf.signal.irfft2d(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.
Computes the inverse 2-dimensional discrete Fourier transform of a real-valued signal over the inner-most 2 dimensions of `input`.
The inner-most 2 dimensions of `input` are assumed to be the result of `RFFT2D`: The inner-most dimension contains the `fft_length / 2 + 1` unique components of the DFT of a real-valued signal. If `fft_length` is not provided, it is computed from the size of the inner-most 2 dimensions of `input`. If the FFT length used to compute `input` is odd, it should be provided since it cannot be inferred properly.
Along each axis `IRFFT2D` is computed on, if `fft_length` (or `fft_length / 2 + 1` for the inner-most dimension) is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **`input`**: A `Tensor` of type `complex64`. A complex64 tensor.
- **`fft_length`**: A `Tensor` of type `int32`. An int32 tensor of shape [2]. The FFT length for each dimension.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `float32`.

tf.signal.irfft3d

- Contents
- Aliases:

Inverse 3D real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.irfft3d`
- `tf.compat.v1.spectral.irfft3d`
- `tf.compat.v2.signal.irfft3d`
- `tf.signal.irfft3d`

```
tf.signal.irfft3d(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.
Computes the inverse 3-dimensional discrete Fourier transform of a real-valued signal over the inner-most 3 dimensions of `input`.
The inner-most 3 dimensions of `input` are assumed to be the result of `RFFT3D`: The inner-most dimension contains the `fft_length / 2 + 1` unique components of the DFT of a real-valued signal. If `fft_length` is not provided, it is computed from the size of the inner-most 3 dimensions of `input`. If the FFT length used to compute `input` is odd, it should be provided since it cannot be inferred properly.

Along each axis `IRFFT3D` is computed on, if `fft_length` (or `fft_length / 2 + 1` for the inner-most dimension) is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **`input`**: A `Tensor` of type `complex64`. A complex64 tensor.
- **`fft_length`**: A `Tensor` of type `int32`. An int32 tensor of shape [3]. The FFT length for each dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

tf.signal.linear_to_mel_weight_matrix

- Contents
- Aliases:

Returns a matrix to warp linear scale spectrograms to the mel scale.

Aliases:

- `tf.compat.v1.signal.linear_to_mel_weight_matrix`
- `tf.compat.v2.signal.linear_to_mel_weight_matrix`
- `tf.signal.linear_to_mel_weight_matrix`

```
tf.signal.linear_to_mel_weight_matrix(
    num_mel_bins=20,
    num_spectrogram_bins=129,
    sample_rate=8000,
    lower_edge_hertz=125.0,
    upper_edge_hertz=3800.0,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in `python/ops/signal/mel_ops.py`.

Returns a weight matrix that can be used to re-weight a `Tensor` containing `num_spectrogram_bins`linearly sampled frequency information from `[0, sample_rate / 2]` into `num_mel_bins` frequency information from `[lower_edge_hertz, upper_edge_hertz]` on the mel scale.

For example, the returned matrix `A` can be used to right-multiply a spectrogram `S` of shape `[frames, num_spectrogram_bins]` of linear scale spectrum values (e.g. STFT magnitudes) to generate a "mel spectrogram" `M` of shape `[frames, num_mel_bins]`.

```
# `S` has shape [frames, num_spectrogram_bins]
# `M` has shape [frames, num_mel_bins]
M = tf.matmul(S, A)
```

The matrix can be used with `tf.tensordot` to convert an arbitrary rank `Tensor` of linear-scale spectral bins into the mel scale.

```
# S has shape [..., num_spectrogram_bins].
# M has shape [..., num_mel_bins].
M = tf.tensordot(S, A, 1)
# tf.tensordot does not support shape inference for this case yet.
```

```
M.set_shape(S.shape[:-1].concatenate(A.shape[-1:]))
```

Args:

- **`num_mel_bins`**: Python int. How many bands in the resulting mel spectrum.
- **`num_spectrogram_bins`**: An integer `Tensor`. How many bins there are in the source spectrogram data, which is understood to be `fft_size // 2 + 1`, i.e. the spectrogram only contains the nonredundant FFT bins.
- **`sample_rate`**: Python float. Samples per second of the input signal used to create the spectrogram. We need this to figure out the actual frequencies for each spectrogram bin, which dictates how they are mapped into the mel scale.
- **`lower_edge_hertz`**: Python float. Lower bound on the frequencies to be included in the mel spectrum. This corresponds to the lower edge of the lowest triangular band.
- **`upper_edge_hertz`**: Python float. The desired top edge of the highest frequency band.
- **`dtype`**: The `DType` of the result matrix. Must be a floating point type.
- **`name`**: An optional name for the operation.

Returns:

A `Tensor` of shape `[num_spectrogram_bins, num_mel_bins]`.

Raises:

- **`ValueError`**: If `num_mel_bins`/`num_spectrogram_bins`/`sample_rate` are not positive, `lower_edge_hertz` is negative, frequency edges are incorrectly ordered, or `upper_edge_hertz`is larger than the Nyquist frequency.

tf.signal.mfccs_from_log_mel_spectrograms

- Contents
- Aliases:

Computes MFCCs of `log_mel_spectrograms`.

Aliases:

- `tf.compat.v1.signal.mfccs_from_log_mel_spectrograms`
- `tf.compat.v2.signal.mfccs_from_log_mel_spectrograms`
- `tf.signal.mfccs_from_log_mel_spectrograms`

```
tf.signal.mfccs_from_log_mel_spectrograms(
    log_mel_spectrograms,
    name=None
)
```

Defined in `python/ops/signal/mfcc_ops.py`.

Implemented with GPU-compatible ops and supports gradients.

Mel-Frequency Cepstral Coefficient (MFCC) calculation consists of taking the DCT-II of a log-magnitude mel-scale spectrogram. HTK's MFCCs use a particular scaling of the DCT-II which is almost orthogonal normalization. We follow this convention.

All `num_mel_bins` MFCCs are returned and it is up to the caller to select a subset of the MFCCs based on their application. For example, it is typical to only use the first few for speech recognition, as this results in an approximately pitch-invariant representation of the signal.

For example:

```
sample_rate = 16000.0
# A Tensor of [batch_size, num_samples] mono PCM samples in the range [-1, 1].
pcm = tf.compat.v1.placeholder(tf.float32, [None, None])
```

```
# A 1024-point STFT with frames of 64 ms and 75% overlap.
stfts = tf.signal.stft(pcm, frame_length=1024, frame_step=256,
                       fft_length=1024)
spectrograms = tf.abs(stfts)

# Warp the linear scale spectrograms into the mel-scale.
num_spectrogram_bins = stfts.shape[-1].value
lower_edge_hertz, upper_edge_hertz, num_mel_bins = 80.0, 7600.0, 80
linear_to_mel_weight_matrix = tf.signal.linear_to_mel_weight_matrix(
  num_mel_bins, num_spectrogram_bins, sample_rate, lower_edge_hertz,
  upper_edge_hertz)
mel_spectrograms = tf.tensordot(
  spectrograms, linear_to_mel_weight_matrix, 1)
mel_spectrograms.set_shape(spectrograms.shape[:-1].concatenate(
  linear_to_mel_weight_matrix.shape[-1:]))

# Compute a stabilized log to get log-magnitude mel-scale spectrograms.
log_mel_spectrograms = tf.math.log(mel_spectrograms + 1e-6)

# Compute MFCCs from log_mel_spectrograms and take the first 13.
mfccs = tf.signal.mfccs_from_log_mel_spectrograms(
  log_mel_spectrograms)[..., :13]
```

Args:

- **`log_mel_spectrograms`**: A `[..., num_mel_bins]` `float32` `Tensor` of log-magnitude mel-scale spectrograms.
- **`name`**: An optional name for the operation.

Returns:

A `[..., num_mel_bins]` `float32` `Tensor` of the MFCCs of `log_mel_spectrograms`.

Raises:

- **`ValueError`**: If `num_mel_bins` is not positive.

tf.signal.overlap_and_add

- Contents
- Aliases:

Reconstructs a signal from a framed representation.

Aliases:

- `tf.compat.v1.signal.overlap_and_add`
- `tf.compat.v2.signal.overlap_and_add`
- `tf.signal.overlap_and_add`

```
tf.signal.overlap_and_add(
    signal,
    frame_step,
    name=None
)
```

Defined in `python/ops/signal/reconstruction_ops.py`.

Adds potentially overlapping frames of a signal with shape `[..., frames, frame_length]`, offsetting subsequent frames by `frame_step`. The resulting tensor has shape `[..., output_size]` where

```
output_size = (frames - 1) * frame_step + frame_length
```

Args:

- **`signal`**: A [..., frames, frame_length] `Tensor`. All dimensions may be unknown, and rank must be at least 2.
- **`frame_step`**: An integer or scalar `Tensor` denoting overlap offsets. Must be less than or equal to `frame_length`.
- **`name`**: An optional name for the operation.

Returns:

A `Tensor` with shape `[..., output_size]` containing the overlap-added frames of `signal`'s inner-most two dimensions.

Raises:

- **`ValueError`**: If `signal`'s rank is less than 2, or `frame_step` is not a scalar integer.

tf.signal.rfft

- Contents
- Aliases:

Real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.rfft`
- `tf.compat.v1.spectral.rfft`
- `tf.compat.v2.signal.rfft`
- `tf.signal.rfft`

```
tf.signal.rfft(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.

Computes the 1-dimensional discrete Fourier transform of a real-valued signal over the inner-most dimension of `input`.

Since the DFT of a real signal is Hermitian-symmetric, `RFFT` only returns the `fft_length / 2 + 1` unique components of the FFT: the zero-frequency term, followed by the `fft_length / 2` positive-frequency terms.

Along the axis `RFFT` is computed on, if `fft_length` is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **`input`**: A `Tensor` of type `float32`. A float32 tensor.
- **`fft_length`**: A `Tensor` of type `int32`. An int32 tensor of shape [1]. The FFT length.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`.

tf.signal.rfft2d

- Contents
- Aliases:

2D real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.rfft2d`
- `tf.compat.v1.spectral.rfft2d`
- `tf.compat.v2.signal.rfft2d`
- `tf.signal.rfft2d`

```
tf.signal.rfft2d(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.

Computes the 2-dimensional discrete Fourier transform of a real-valued signal over the inner-most 2 dimensions of `input`.

Since the DFT of a real signal is Hermitian-symmetric, `RFFT2D` only returns the `fft_length / 2 + 1`unique components of the FFT for the inner-most dimension of `output`: the zero-frequency term, followed by the `fft_length / 2` positive-frequency terms.

Along each axis `RFFT2D` is computed on, if `fft_length` is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **`input`**: A `Tensor` of type `float32`. A float32 tensor.
- **`fft_length`**: A `Tensor` of type `int32`. An int32 tensor of shape [2]. The FFT length for each dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`.

tf.signal.rfft3d

- Contents
- Aliases:

3D real-valued fast Fourier transform.

Aliases:

- `tf.compat.v1.signal.rfft3d`
- `tf.compat.v1.spectral.rfft3d`
- `tf.compat.v2.signal.rfft3d`
- `tf.signal.rfft3d`

```
tf.signal.rfft3d(
    input_tensor,
    fft_length=None,
    name=None
)
```

Defined in `python/ops/signal/fft_ops.py`.

Computes the 3-dimensional discrete Fourier transform of a real-valued signal over the inner-most 3 dimensions of `input`.

Since the DFT of a real signal is Hermitian-symmetric, `RFFT3D` only returns the `fft_length / 2 + 1`unique components of the FFT for the inner-most dimension of `output`: the zero-frequency term, followed by the `fft_length / 2` positive-frequency terms.
Along each axis `RFFT3D` is computed on, if `fft_length` is smaller than the corresponding dimension of `input`, the dimension is cropped. If it is larger, the dimension is padded with zeros.

Args:

- **`input`**: A `Tensor` of type `float32`. A float32 tensor.
- **`fft_length`**: A `Tensor` of type `int32`. An int32 tensor of shape [3]. The FFT length for each dimension.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `complex64`.

tf.signal.stft

- Contents
- Aliases:

Computes the Short-time Fourier Transform of `signals`.

Aliases:

- `tf.compat.v1.signal.stft`
- `tf.compat.v2.signal.stft`
- `tf.signal.stft`

```
tf.signal.stft(
    signals,
    frame_length,
    frame_step,
    fft_length=None,
    window_fn=tf.signal.hann_window,
    pad_end=False,
    name=None
)
```

Defined in `python/ops/signal/spectral_ops.py`.
Implemented with GPU-compatible ops and supports gradients.

Args:

- **`signals`**: A `[..., samples]` `float32` `Tensor` of real-valued signals.
- **`frame_length`**: An integer scalar `Tensor`. The window length in samples.
- **`frame_step`**: An integer scalar `Tensor`. The number of samples to step.
- **`fft_length`**: An integer scalar `Tensor`. The size of the FFT to apply. If not provided, uses the smallest power of 2 enclosing `frame_length`.
- **`window_fn`**: A callable that takes a window length and a `dtype` keyword argument and returns a `[window_length]` `Tensor` of samples in the provided datatype. If set to `None`, no windowing is used.
- **`pad_end`**: Whether to pad the end of `signals` with zeros when the provided frame length and step produces a frame that lies partially past its end.
- **`name`**: An optional name for the operation.

Returns:

A `[..., frames, fft_unique_bins]` `Tensor` of `complex64` STFT values where`fft_unique_bins` is `fft_length // 2 + 1` (the unique components of the FFT).

Raises:

- `ValueError`: If `signals` is not at least rank 1, `frame_length` is not scalar, or `frame_step` is not scalar.

Module: tf.sparse

- **Contents**
- Classes
- Functions

Sparse Tensor Representation.
See also `tf.SparseTensor`.

Classes

`class SparseTensor`: Represents a sparse tensor.

Functions

`add(...)`: Adds two tensors, at least one of each is a `SparseTensor`.
`concat(...)`: Concatenates a list of `SparseTensor` along the specified dimension. (deprecated arguments)
`cross(...)`: Generates sparse cross from a list of sparse and dense tensors.
`cross_hashed(...)`: Generates hashed sparse cross from a list of sparse and dense tensors.
`expand_dims(...)`: Inserts a dimension of 1 into a tensor's shape.
`eye(...)`: Creates a two-dimensional sparse tensor with ones along the diagonal.
`fill_empty_rows(...)`: Fills empty rows in the input 2-D `SparseTensor` with a default value.
`mask(...)`: Masks elements of `IndexedSlices`.
`maximum(...)`: Returns the element-wise max of two SparseTensors.
`minimum(...)`: Returns the element-wise min of two SparseTensors.
`reduce_max(...)`: Computes the max of elements across dimensions of a SparseTensor.
`reduce_sum(...)`: Computes the sum of elements across dimensions of a SparseTensor.
`reorder(...)`: Reorders a `SparseTensor` into the canonical, row-major ordering.
`reset_shape(...)`: Resets the shape of a `SparseTensor` with indices and values unchanged.
`reshape(...)`: Reshapes a `SparseTensor` to represent values in a new dense shape.
`retain(...)`: Retains specified non-empty values within a `SparseTensor`.
`segment_mean(...)`: Computes the mean along sparse segments of a tensor.
`segment_sqrt_n(...)`: Computes the sum along sparse segments of a tensor divided by the sqrt(N).
`segment_sum(...)`: Computes the sum along sparse segments of a tensor.
`slice(...)`: Slice a `SparseTensor` based on the `start` and `size.
`softmax(...)`: Applies softmax to a batched N-D `SparseTensor`.
`sparse_dense_matmul(...)`: Multiply SparseTensor (of rank 2) "A" by dense matrix "B".
`split(...)`: Split a `SparseTensor` into `num_split` tensors along `axis`.
`to_dense(...)`: Converts a `SparseTensor` into a dense tensor.
`to_indicator(...)`: Converts a `SparseTensor` of ids into a dense bool indicator tensor.
`transpose(...)`: Transposes a `SparseTensor`

tf.sparse.add

- Contents
- Aliases:

Adds two tensors, at least one of each is a `SparseTensor`.

Aliases:

- `tf.compat.v2.sparse.add`

- `tf.sparse.add`

```
tf.sparse.add(
    a,
    b,
    threshold=0
)
```

Defined in `python/ops/sparse_ops.py`.
If one `SparseTensor` and one `Tensor` are passed in, returns a `Tensor`. If both arguments are `SparseTensor`s, this returns a `SparseTensor`. The order of arguments does not matter. Use vanilla `tf.add()` for adding two dense `Tensor`s.
The shapes of the two operands must match: broadcasting is not supported.
The indices of any input `SparseTensor` are assumed ordered in standard lexicographic order. If this is not the case, before this step run `SparseReorder` to restore index ordering.
If both arguments are sparse, we perform "clipping" as follows. By default, if two values sum to zero at some index, the output `SparseTensor` would still include that particular location in its index, storing a zero in the corresponding value slot. To override this, callers can specify `threshold`, indicating that if the sum has a magnitude strictly smaller than `threshold`, its corresponding value and index would then not be included. In particular, `threshold == 0.0` (default) means everything is kept and actual thresholding happens only for a positive value.
For example, suppose the logical sum of two sparse operands is (densified):

```
[       2]
[.1     0]
[ 6   -.2]
```

Then,

- `threshold == 0` (the default): all 5 index/value pairs will be returned.
- `threshold == 0.11`: only .1 and 0 will vanish, and the remaining three index/value pairs will be returned.
- `threshold == 0.21`: .1, 0, and -.2 will vanish.

Args:

- **`a`**: The first operand; `SparseTensor` or `Tensor`.
- **`b`**: The second operand; `SparseTensor` or `Tensor`. At least one operand must be sparse.
- **`threshold`**: A 0-D `Tensor`. The magnitude threshold that determines if an output value/index pair takes space. Its dtype should match that of the values if they are real; if the latter are complex64/complex128, then the dtype should be float32/float64, correspondingly.

Returns:

A `SparseTensor` or a `Tensor`, representing the sum.

Raises:

- **`TypeError`**: If both `a` and `b` are `Tensor`s. Use `tf.add()` instead.

tf.sparse.concat

- Contents
- Aliases:
- Used in the guide:

Concatenates a list of `SparseTensor` along the specified dimension. (deprecated arguments)

Aliases:

- `tf.compat.v2.sparse.concat`
- `tf.sparse.concat`

```
tf.sparse.concat(
    axis,
    sp_inputs,
    expand_nonconcat_dims=False,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

Used in the guide:

- Ragged Tensors

Warning: SOME ARGUMENTS ARE DEPRECATED: `(concat_dim)`. They will be removed in a future version. Instructions for updating: concat_dim is deprecated, use axis instead

Concatenation is with respect to the dense versions of each sparse input. It is assumed that each inputs is a `SparseTensor` whose elements are ordered along increasing dimension number.

If expand_nonconcat_dim is False, all inputs' shapes must match, except for the concat dimension. If expand_nonconcat_dim is True, then inputs' shapes are allowed to vary among all inputs.

The `indices`, `values`, and `shapes` lists must have the same length.

If expand_nonconcat_dim is False, then the output shape is identical to the inputs', except along the concat dimension, where it is the sum of the inputs' sizes along that dimension.

If expand_nonconcat_dim is True, then the output shape along the non-concat dimensions will be expand to be the largest among all inputs, and it is the sum of the inputs sizes along the concat dimension.

The output elements will be resorted to preserve the sort order along increasing dimension number.

This op runs in `O(M log M)` time, where `M` is the total number of non-empty values across all inputs. This is due to the need for an internal sort in order to concatenate efficiently across an arbitrary dimension.

For example, if `axis = 1` and the inputs are

```
sp_inputs[0]: shape = [2, 3]
[0, 2]: "a"
[1, 0]: "b"
[1, 1]: "c"

sp_inputs[1]: shape = [2, 4]
[0, 1]: "d"
[0, 2]: "e"
```

then the output will be

```
shape = [2, 7]
[0, 2]: "a"
[0, 4]: "d"
[0, 5]: "e"
[1, 0]: "b"
[1, 1]: "c"
```

Graphically this is equivalent to doing

```
[    a] concat [  d e  ] = [     a   d e  ]
[b c  ]        [       ]   [b c           ]
```

Another example, if 'axis = 1' and the inputs are

```
sp_inputs[0]: shape = [3, 3]
[0, 2]: "a"
[1, 0]: "b"
[2, 1]: "c"

sp_inputs[1]: shape = [2, 4]
[0, 1]: "d"
[0, 2]: "e"
```

if expand_nonconcat_dim = False, this will result in an error. But if expand_nonconcat_dim = True, this will result in:

```
shape = [3, 7]
[0, 2]: "a"
[0, 4]: "d"
[0, 5]: "e"
[1, 0]: "b"
[2, 1]: "c"
```

Graphically this is equivalent to doing

```
[    a] concat [  d e  ] = [     a   d e  ]
[b    ]        [       ]   [b             ]
[  c  ]                    [  c           ]
```

Args:

- **axis**: Dimension to concatenate along. Must be in range [-rank, rank), where rank is the number of dimensions in each input `SparseTensor`.
- **sp_inputs**: List of `SparseTensor` to concatenate.
- **name**: A name prefix for the returned tensors (optional).
- **expand_nonconcat_dim**: Whether to allow the expansion in the non-concat dimensions. Defaulted to False.
- **concat_dim**: The old (deprecated) name for axis.
- **expand_nonconcat_dims**: alias for expand_nonconcat_dim

Returns:

A `SparseTensor` with the concatenated output.

Raises:

- **TypeError**: If `sp_inputs` is not a list of `SparseTensor`.

tf.sparse.cross

- Contents
- Aliases:

Generates sparse cross from a list of sparse and dense tensors.

Aliases:

- `tf.compat.v1.sparse.cross`
- `tf.compat.v2.sparse.cross`
- `tf.sparse.cross`

```
tf.sparse.cross(
    inputs,
    name=None
```

```
)
```

Defined in `python/ops/sparse_ops.py`.

For example, if the inputs are

```
* inputs[0]: SparseTensor with shape = [2, 2]
  [0, 0]: "a"
  [1, 0]: "b"
  [1, 1]: "c"
* inputs[1]: SparseTensor with shape = [2, 1]
  [0, 0]: "d"
  [1, 0]: "e"
* inputs[2]: Tensor [["f"], ["g"]]
```

then the output will be:

```
shape = [2, 2]
[0, 0]: "a_X_d_X_f"
[1, 0]: "b_X_e_X_g"
[1, 1]: "c_X_e_X_g"
```

Args:

- **`inputs`**: An iterable of `Tensor` or `SparseTensor`.
- **`name`**: Optional name for the op.

Returns:

A `SparseTensor` of type `string`.

tf.sparse.cross_hashed

- Contents
- Aliases:

Generates hashed sparse cross from a list of sparse and dense tensors.

Aliases:

- `tf.compat.v1.sparse.cross_hashed`
- `tf.compat.v2.sparse.cross_hashed`
- `tf.sparse.cross_hashed`

```
tf.sparse.cross_hashed(
    inputs,
    num_buckets=0,
    hash_key=None,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

For example, if the inputs are

```
* inputs[0]: SparseTensor with shape = [2, 2]
  [0, 0]: "a"
  [1, 0]: "b"
  [1, 1]: "c"
* inputs[1]: SparseTensor with shape = [2, 1]
  [0, 0]: "d"
```

```
  [1, 0]: "e"
* inputs[2]: Tensor [["f"], ["g"]]
```

then the output will be:

```
shape = [2, 2]
[0, 0]: FingerprintCat64(
            Fingerprint64("f"), FingerprintCat64(
                Fingerprint64("d"), Fingerprint64("a")))
[1, 0]: FingerprintCat64(
            Fingerprint64("g"), FingerprintCat64(
                Fingerprint64("e"), Fingerprint64("b")))
[1, 1]: FingerprintCat64(
            Fingerprint64("g"), FingerprintCat64(
                Fingerprint64("e"), Fingerprint64("c")))
```

Args:

- **`inputs`**: An iterable of `Tensor` or `SparseTensor`.
- **`num_buckets`**: An `int` that is `>= 0`. output = hashed_value%num_buckets if num_buckets > 0 else hashed_value.
- **`hash_key`**: Integer hash_key that will be used by the `FingerprintCat64` function. If not given, will use a default key.
- **`name`**: Optional name for the op.

Returns:

A `SparseTensor` of type `int64`.

tf.sparse.expand_dims

- Contents
- Aliases:

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

Aliases:

- `tf.compat.v1.sparse.expand_dims`
- `tf.compat.v2.sparse.expand_dims`
- `tf.sparse.expand_dims`

```
tf.sparse.expand_dims(
    sp_input,
    axis=None,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

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

Args:

- **`sp_input`**: A `SparseTensor`.
- **`axis`**: 0-D (scalar). Specifies the dimension index at which to expand the shape of `input`. Must be in the range `[-rank(sp_input) - 1, rank(sp_input)]`.
- **`name`**: The name of the output `SparseTensor`.

Returns:
A `SparseTensor` with the same data as `sp_input`, but its shape has an additional dimension of size 1 added.

tf.sparse.eye

- Contents
- Aliases:

Creates a two-dimensional sparse tensor with ones along the diagonal.

Aliases:

- `tf.compat.v1.sparse.eye`
- `tf.compat.v2.sparse.eye`
- `tf.sparse.eye`

```
tf.sparse.eye(
    num_rows,
    num_columns=None,
    dtype=tf.dtypes.float32,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

Args:

- **`num_rows`**: Non-negative integer or `int32` scalar `tensor` giving the number of rows in the resulting matrix.
- **`num_columns`**: Optional non-negative integer or `int32` scalar `tensor` giving the number of columns in the resulting matrix. Defaults to `num_rows`.
- **`dtype`**: The type of element in the resulting `Tensor`.
- **`name`**: A name for this `Op`. Defaults to "eye".

Returns:
A `SparseTensor` of shape [num_rows, num_columns] with ones along the diagonal.

tf.sparse.fill_empty_rows

- Contents
- Aliases:

Fills empty rows in the input 2-D `SparseTensor` with a default value.

Aliases:

- `tf.compat.v1.sparse.fill_empty_rows`
- `tf.compat.v1.sparse_fill_empty_rows`
- `tf.compat.v2.sparse.fill_empty_rows`
- `tf.sparse.fill_empty_rows`

```
tf.sparse.fill_empty_rows(
    sp_input,
    default_value,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

This op adds entries with the specified `default_value` at index `[row, 0]` for any row in the input that does not already have a value.
For example, suppose `sp_input` has shape `[5, 6]` and non-empty values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

Rows 1 and 4 are empty, so the output will be of shape `[5, 6]` with values:

```
[0, 1]: a
[0, 3]: b
[1, 0]: default_value
[2, 0]: c
[3, 1]: d
[4, 0]: default_value
```

Note that the input may have empty columns at the end, with no effect on this op.
The output `SparseTensor` will be in row-major order and will have the same shape as the input.
This op also returns an indicator vector such that

```
empty_row_indicator[i] = True iff row i was an empty row.
```

Args:

- **`sp_input`**: A `SparseTensor` with shape `[N, M]`.
- **`default_value`**: The value to fill for empty rows, with the same type as `sp_input.`
- **`name`**: A name prefix for the returned tensors (optional)

Returns:

- **`sp_ordered_output`**: A `SparseTensor` with shape `[N, M]`, and with all empty rows filled in with `default_value`.
- **`empty_row_indicator`**: A bool vector of length `N` indicating whether each input row was empty.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.mask

- Contents
- Aliases:

Masks elements of `IndexedSlices`.

Aliases:

- `tf.compat.v1.sparse.mask`
- `tf.compat.v1.sparse_mask`
- `tf.compat.v2.sparse.mask`
- `tf.sparse.mask`

```
tf.sparse.mask(
    a,
    mask_indices,
    name=None
)
```

Defined in `python/ops/array_ops.py`.

Given an `IndexedSlices` instance `a`, returns another `IndexedSlices` that contains a subset of the slices of `a`. Only the slices at indices not specified in `mask_indices` are returned.
This is useful when you need to extract a subset of slices in an `IndexedSlices` object.

For example:

```
# `a` contains slices at indices [12, 26, 37, 45] from a large tensor
# with shape [1000, 10]
a.indices  # [12, 26, 37, 45]
tf.shape(a.values)  # [4, 10]

# `b` will be the subset of `a` slices at its second and third indices, so
# we want to mask its first and last indices (which are at absolute
# indices 12, 45)
b = tf.sparse.mask(a, [12, 45])

b.indices  # [26, 37]
tf.shape(b.values)  # [2, 10]
```

Args:

- **`a`**: An `IndexedSlices` instance.
- **`mask_indices`**: Indices of elements to mask.
- **`name`**: A name for the operation (optional).

Returns:
The masked `IndexedSlices` instance.

tf.sparse.maximum

- Contents
- Aliases:

Returns the element-wise max of two SparseTensors.

Aliases:

- `tf.compat.v1.sparse.maximum`
- `tf.compat.v1.sparse_maximum`
- `tf.compat.v2.sparse.maximum`
- `tf.sparse.maximum`

```
tf.sparse.maximum(
    sp_a,
    sp_b,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
Assumes the two SparseTensors have the same shape, i.e., no broadcasting. Example:

```
sp_zero = sparse_tensor.SparseTensor([[0]], [0], [7])
sp_one = sparse_tensor.SparseTensor([[1]], [1], [7])
res = tf.sparse.maximum(sp_zero, sp_one).eval()
# "res" should be equal to SparseTensor([[0], [1]], [0, 1], [7]).
```

Args:

- **`sp_a`**: a `SparseTensor` operand whose dtype is real, and indices lexicographically ordered.
- **`sp_b`**: the other `SparseTensor` operand with the same requirements (and the same shape).
- **`name`**: optional name of the operation.

Returns:

- **`output`**: the output SparseTensor.

tf.sparse.minimum

- Contents
- Aliases:

Returns the element-wise min of two SparseTensors.

Aliases:

- `tf.compat.v1.sparse.minimum`
- `tf.compat.v1.sparse_minimum`
- `tf.compat.v2.sparse.minimum`
- `tf.sparse.minimum`

```
tf.sparse.minimum(
    sp_a,
    sp_b,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

Assumes the two SparseTensors have the same shape, i.e., no broadcasting. Example:

```
sp_zero = sparse_tensor.SparseTensor([[0]], [0], [7])
sp_one = sparse_tensor.SparseTensor([[1]], [1], [7])
res = tf.sparse.minimum(sp_zero, sp_one).eval()
# "res" should be equal to SparseTensor([[0], [1]], [0, 0], [7]).
```

Args:

- **`sp_a`**: a `SparseTensor` operand whose dtype is real, and indices lexicographically ordered.
- **`sp_b`**: the other `SparseTensor` operand with the same requirements (and the same shape).
- **`name`**: optional name of the operation.

Returns:

- **`output`**: the output SparseTensor.

tf.sparse.reduce_max

- Contents
- Aliases:

Computes the max of elements across dimensions of a SparseTensor.

Aliases:

- `tf.compat.v2.sparse.reduce_max`
- `tf.sparse.reduce_max`

```
tf.sparse.reduce_max(
    sp_input,
    axis=None,
    keepdims=None,
```

```
    output_is_sparse=False,
    name=None
)
```

Defined in python/ops/sparse_ops.py.
This Op takes a SparseTensor and is the sparse counterpart to `tf.reduce_max()`. In particular, this Op also returns a dense `Tensor` if `output_is_sparse` is `False`, or a `SparseTensor` if `output_is_sparse` is `True`.
Note: A gradient is not defined for this function, so it can't be used in training models that need gradient descent.
Reduces `sp_input` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` has no entries, all dimensions are reduced, and a tensor with a single element is returned. Additionally, the axes can be negative, similar to the indexing rules in Python.
The values not defined in `sp_input` don't participate in the reduce max, as opposed to be implicitly assumed 0 -- hence it can return negative values for sparse `axis`. But, in case there are no values in`axis`, it will reduce to 0. See second example below.

For example:

```
# 'x' represents [[1, ?, 2]
#                 [?, 3, ?]]
# where ? is implicitly-zero.
tf.sparse.reduce_max(x) ==> 3
tf.sparse.reduce_max(x, 0) ==> [1, 3, 2]
tf.sparse.reduce_max(x, 1) ==> [2, 3]  # Can also use -1 as the axis.
tf.sparse.reduce_max(x, 1, keepdims=True) ==> [[2], [3]]
tf.sparse.reduce_max(x, [0, 1]) ==> 3

# 'y' represents [[-7, ?]
#                 [ 4, 3]
#                 [ ?, ?]
tf.sparse.reduce_max(x, 1) ==> [-7, 4, 0]
```

Args:

- **`sp_input`**: The SparseTensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce; list or scalar. If `None` (the default), reduces all dimensions.
- **`keepdims`**: If true, retain reduced dimensions with length 1.
- **`output_is_sparse`**: If true, returns a `SparseTensor` instead of a dense `Tensor` (the default).
- **`name`**: A name for the operation (optional).

Returns:

The reduced Tensor or the reduced SparseTensor if `output_is_sparse` is True.

tf.sparse.reduce_sum

- Contents
- Aliases:

Computes the sum of elements across dimensions of a SparseTensor.

Aliases:

- `tf.compat.v2.sparse.reduce_sum`

- `tf.sparse.reduce_sum`

```
tf.sparse.reduce_sum(
    sp_input,
    axis=None,
    keepdims=None,
    output_is_sparse=False,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
This Op takes a SparseTensor and is the sparse counterpart to `tf.reduce_sum()`. In particular, this Op also returns a dense `Tensor` if `output_is_sparse` is `False`, or a `SparseTensor` if `output_is_sparse` is `True`.
Note: if **`output_is_sparse`** is True, a gradient is not defined for this function, so it can't be used in training models that need gradient descent.
Reduces `sp_input` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` has no entries, all dimensions are reduced, and a tensor with a single element is returned. Additionally, the axes can be negative, similar to the indexing rules in Python.

For example:

```
# 'x' represents [[1, ?, 1]
#                 [?, 1, ?]]
# where ? is implicitly-zero.
tf.sparse.reduce_sum(x) ==> 3
tf.sparse.reduce_sum(x, 0) ==> [1, 1, 1]
tf.sparse.reduce_sum(x, 1) ==> [2, 1]  # Can also use -1 as the axis.
tf.sparse.reduce_sum(x, 1, keepdims=True) ==> [[2], [1]]
tf.sparse.reduce_sum(x, [0, 1]) ==> 3
```

Args:

- **`sp_input`**: The SparseTensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce; list or scalar. If `None` (the default), reduces all dimensions.
- **`keepdims`**: If true, retain reduced dimensions with length 1.
- **`output_is_sparse`**: If true, returns a `SparseTensor` instead of a dense `Tensor` (the default).
- **`name`**: A name for the operation (optional).

Returns:
The reduced Tensor or the reduced SparseTensor if `output_is_sparse` is True.

tf.sparse.reorder

- Contents
- Aliases:

Reorders a `SparseTensor` into the canonical, row-major ordering.

Aliases:

- `tf.compat.v1.sparse.reorder`
- `tf.compat.v1.sparse_reorder`
- `tf.compat.v2.sparse.reorder`
- `tf.sparse.reorder`

```
tf.sparse.reorder(
    sp_input,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
Note that by convention, all sparse ops preserve the canonical ordering along increasing dimension number. The only time ordering can be violated is during manual manipulation of the indices and values to add entries.
Reordering does not affect the shape of the `SparseTensor`.
For example, if `sp_input` has shape `[4, 5]` and `indices` / `values`:

```
[0, 3]: b
[0, 1]: a
[3, 1]: d
[2, 0]: c
```

then the output will be a `SparseTensor` of shape `[4, 5]` and `indices` / `values`:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

Args:

- **`sp_input`**: The input `SparseTensor`.
- **`name`**: A name prefix for the returned tensors (optional)

Returns:
A `SparseTensor` with the same shape and non-empty values, but in canonical ordering.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.reset_shape

- Contents
- Aliases:

Resets the shape of a `SparseTensor` with indices and values unchanged.

Aliases:

- `tf.compat.v1.sparse.reset_shape`
- `tf.compat.v1.sparse_reset_shape`
- `tf.compat.v2.sparse.reset_shape`
- `tf.sparse.reset_shape`

```
tf.sparse.reset_shape(
    sp_input,
    new_shape=None
)
```

Defined in `python/ops/sparse_ops.py`.
If `new_shape` is None, returns a copy of `sp_input` with its shape reset to the tight bounding box of `sp_input`. This will be a shape consisting of all zeros if sp_input has no values.

If `new_shape` is provided, then it must be larger or equal in all dimensions compared to the shape of `sp_input`. When this condition is met, the returned SparseTensor will have its shape reset to `new_shape` and its indices and values unchanged from that of `sp_input`.

For example:
Consider a `sp_input` with shape [2, 3, 5]:

- It is an error to set `new_shape` as [3, 7] since this represents a rank-2 tensor while `sp_input` is rank-3. This is either a ValueError during graph construction (if both shapes are known) or an OpError during run time.
- Setting `new_shape` as [2, 3, 6] will be fine as this shape is larger or equal in every dimension compared to the original shape [2, 3, 5].
- On the other hand, setting new_shape as [2, 3, 4] is also an error: The third dimension is smaller than the original shape [2, 3, 5](https://www.tensorflow.org/api_docs/python/tf/sparse/reset_shape).
- If `new_shape` is None, the returned SparseTensor will have a shape [2, 3, 4], which is the tight bounding box of `sp_input`.

Args:

- **`sp_input`**: The input `SparseTensor`.
- **`new_shape`**: None or a vector representing the new shape for the returned `SparseTensor`.

Returns:
A `SparseTensor` indices and values unchanged from `input_sp`. Its shape is `new_shape` if that is set. Otherwise it is the tight bounding box of `input_sp`

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.
- **`ValueError`**: If `new_shape` represents a tensor with a different rank from that of `sp_input` (if shapes are known when graph is constructed).
- **`ValueError`**: If `new_shape` is determined during graph build to have dimension sizes that are too small.
- **`OpError`**: - If `new_shape` has dimension sizes that are too small.
- If shapes are not known during graph construction time, and during run time it is found out that the ranks do not match.

tf.sparse.reshape

- Contents
- Aliases:

Reshapes a `SparseTensor` to represent values in a new dense shape.

Aliases:

- `tf.compat.v1.sparse.reshape`
- `tf.compat.v1.sparse_reshape`
- `tf.compat.v2.sparse.reshape`
- `tf.sparse.reshape`

```
tf.sparse.reshape(
    sp_input,
    shape,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
This operation has the same semantics as `reshape` on the represented dense tensor. The indices of non-empty values in `sp_input` are recomputed based on the new dense shape, and a

new `SparseTensor` is returned containing the new indices and new shape. The order of non-empty values in `sp_input` is unchanged.
If one component of `shape` is the special value -1, the size of that dimension is computed so that the total dense size remains constant. At most one component of `shape` can be -1. The number of dense elements implied by `shape` must be the same as the number of dense elements originally represented by `sp_input`.
For example, if `sp_input` has shape `[2, 3, 6]` and `indices` / `values`:

```
[0, 0, 0]: a
[0, 0, 1]: b
[0, 1, 0]: c
[1, 0, 0]: d
[1, 2, 3]: e
```

and `shape` is `[9, -1]`, then the output will be a `SparseTensor` of shape `[9, 4]` and `indices` / `values`:

```
[0, 0]: a
[0, 1]: b
[1, 2]: c
[4, 2]: d
[8, 1]: e
```

Args:

- **`sp_input`**: The input `SparseTensor`.
- **`shape`**: A 1-D (vector) int64 `Tensor` specifying the new dense shape of the represented `SparseTensor`.
- **`name`**: A name prefix for the returned tensors (optional)

Returns:
A `SparseTensor` with the same non-empty values but with indices calculated by the new dense shape.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.
- **`ValueError`**: If argument `shape` requests a `SparseTensor` with a different number of elements than `sp_input`.
- **`ValueError`**: If `shape` has more than one inferred (== -1) dimension.

tf.sparse.retain

- Contents
- Aliases:

Retains specified non-empty values within a `SparseTensor`.

Aliases:

- `tf.compat.v1.sparse.retain`
- `tf.compat.v1.sparse_retain`
- `tf.compat.v2.sparse.retain`
- `tf.sparse.retain`

```
tf.sparse.retain(
    sp_input,
    to_retain
```

```
)
```

Defined in `python/ops/sparse_ops.py`.
For example, if `sp_input` has shape `[4, 5]` and 4 non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

and `to_retain = [True, False, False, True]`, then the output will be a `SparseTensor` of shape `[4, 5]` with 2 non-empty values:

```
[0, 1]: a
[3, 1]: d
```

Args:

- **`sp_input`**: The input `SparseTensor` with `N` non-empty elements.
- **`to_retain`**: A bool vector of length `N` with `M` true values.

Returns:
A `SparseTensor` with the same shape as the input and `M` non-empty elements corresponding to the true positions in `to_retain`.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.segment_mean

- Contents
- Aliases:

Computes the mean along sparse segments of a tensor.

Aliases:

- `tf.compat.v2.sparse.segment_mean`
- `tf.sparse.segment_mean`

```
tf.sparse.segment_mean(
    data,
    indices,
    segment_ids,
    num_segments=None,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Read the section on segmentation for an explanation of segments.
Like `tf.math.segment_mean`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`. `segment_ids` is allowed to have missing ids, in which case the output will be zeros at those indices. In those cases `num_segments` is used to determine the size of the output.

Args:

- **`data`**: A `Tensor` with data that will be assembled in the output.
- **`indices`**: A 1-D `Tensor` with indices into `data`. Has same rank as `segment_ids`.

- **segment_ids**: A 1-D `Tensor` with indices into the output `Tensor`. Values should be sorted and can be repeated.
- **num_segments**: An optional int32 scalar. Indicates the size of the output `Tensor`.
- **name**: A name for the operation (optional).

Returns:
A `tensor` of the shape as data, except for dimension 0 which has size `k`, the number of segments specified via `num_segments` or inferred for the last element in `segments_ids`.

tf.sparse.segment_sqrt_n

- Contents
- Aliases:

Computes the sum along sparse segments of a tensor divided by the sqrt(N).

Aliases:

- `tf.compat.v2.sparse.segment_sqrt_n`
- `tf.sparse.segment_sqrt_n`

```
tf.sparse.segment_sqrt_n(
    data,
    indices,
    segment_ids,
    num_segments=None,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Read the section on segmentation for an explanation of segments.
Like `tf.sparse.segment_mean`, but instead of dividing by the size of the segment, `N`, divide by `sqrt(N)` instead.

Args:

- **data**: A `Tensor` with data that will be assembled in the output.
- **indices**: A 1-D `Tensor` with indices into `data`. Has same rank as `segment_ids`.
- **segment_ids**: A 1-D `Tensor` with indices into the output `Tensor`. Values should be sorted and can be repeated.
- **num_segments**: An optional int32 scalar. Indicates the size of the output `Tensor`.
- **name**: A name for the operation (optional).

Returns:
A `tensor` of the shape as data, except for dimension 0 which has size `k`, the number of segments specified via `num_segments` or inferred for the last element in `segments_ids`.

tf.sparse.segment_sum

- Contents
- Aliases:

Computes the sum along sparse segments of a tensor.

Aliases:

- `tf.compat.v2.sparse.segment_sum`
- `tf.sparse.segment_sum`

```
tf.sparse.segment_sum(
    data,
```

```
    indices,
    segment_ids,
    num_segments=None,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Read the section on segmentation for an explanation of segments.
Like `tf.math.segment_sum`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`. `segment_ids` is allowed to have missing ids, in which case the output will be zeros at those indices. In those cases `num_segments` is used to determine the size of the output.

For example:

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])

# Select two rows, one segment.
tf.sparse.segment_sum(c, tf.constant([0, 1]), tf.constant([0, 0]))
# => [[0 0 0 0]]

# Select two rows, two segment.
tf.sparse.segment_sum(c, tf.constant([0, 1]), tf.constant([0, 1]))
# => [[ 1  2  3  4]
#     [-1 -2 -3 -4]]

# With missing segment ids.
tf.sparse.segment_sum(c, tf.constant([0, 1]), tf.constant([0, 2]),
                      num_segments=4)
# => [[ 1  2  3  4]
#     [ 0  0  0  0]
#     [-1 -2 -3 -4]
#     [ 0  0  0  0]]

# Select all rows, two segments.
tf.sparse.segment_sum(c, tf.constant([0, 1, 2]), tf.constant([0, 0, 1]))
# => [[0 0 0 0]
#     [5 6 7 8]]

# Which is equivalent to:
tf.math.segment_sum(c, tf.constant([0, 0, 1]))
```

Args:

- **`data`**: A `Tensor` with data that will be assembled in the output.
- **`indices`**: A 1-D `Tensor` with indices into `data`. Has same rank as `segment_ids`.
- **`segment_ids`**: A 1-D `Tensor` with indices into the output `Tensor`. Values should be sorted and can be repeated.
- **`num_segments`**: An optional int32 scalar. Indicates the size of the output `Tensor`.
- **`name`**: A name for the operation (optional).

Returns:
A `tensor` of the shape as data, except for dimension 0 which has size `k`, the number of segments specified via `num_segments` or inferred for the last element in `segments_ids`.

tf.sparse.slice

- Contents
- Aliases:

Slice a `SparseTensor` based on the `start` and `size.

Aliases:

- `tf.compat.v1.sparse.slice`
- `tf.compat.v1.sparse_slice`
- `tf.compat.v2.sparse.slice`
- `tf.sparse.slice`

```
tf.sparse.slice(
    sp_input,
    start,
    size,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
For example, if the input is

```
input_tensor = shape = [2, 7]
[    a   d e  ]
[b c          ]
```

Graphically the output tensors are:

```
sparse.slice([0, 0], [2, 4]) = shape = [2, 4]
[    a  ]
[b c    ]

sparse.slice([0, 4], [2, 3]) = shape = [2, 3]
[ d e  ]
[      ]
```

Args:

- **`sp_input`**: The `SparseTensor` to split.
- **`start`**: 1-D. tensor represents the start of the slice.
- **`size`**: 1-D. tensor represents the size of the slice.
- **`name`**: A name for the operation (optional).

Returns:
A `SparseTensor` objects resulting from splicing.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.softmax

- Contents

- Aliases:

Applies softmax to a batched N-D `SparseTensor`.

Aliases:

- `tf.compat.v1.sparse.softmax`
- `tf.compat.v1.sparse_softmax`
- `tf.compat.v2.sparse.softmax`
- `tf.sparse.softmax`

```
tf.sparse.softmax(
    sp_input,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

The inputs represent an N-D SparseTensor with logical shape `[..., B, C]` (where `N >= 2`), and with indices sorted in the canonical lexicographic order.

This op is equivalent to applying the normal `tf.nn.softmax()` to each innermost logical submatrix with shape `[B, C]`, but with the catch that *the implicitly zero elements do not participate*. Specifically, the algorithm is equivalent to:

(1) Applies `tf.nn.softmax()` to a densified view of each innermost submatrix with shape `[B, C]`, along the size-C dimension; (2) Masks out the original implicitly-zero locations; (3) Renormalizes the remaining elements.

Hence, the `SparseTensor` result has exactly the same non-zero indices and shape.

Example:

```
# First batch:
# [?   e.]
# [1.  ? ]
# Second batch:
# [e   ? ]
# [e   e ]
shape = [2, 2, 2]  # 3-D SparseTensor
values = np.asarray([[[0., np.e], [1., 0.]], [[np.e, 0.], [np.e, np.e]]])
indices = np.vstack(np.where(values)).astype(np.int64).T

result = tf.sparse.softmax(tf.SparseTensor(indices, values, shape))
# ...returning a 3-D SparseTensor, equivalent to:
# [?   1.]     [1    ?]
# [1.  ? ] and [.5  .5]
# where ? means implicitly zero.
```

Args:

- **`sp_input`**: N-D `SparseTensor`, where `N >= 2`.
- **`name`**: optional name of the operation.

Returns:

- **`output`**: N-D `SparseTensor` representing the results.

tf.sparse.SparseTensor

- Contents
- Class SparseTensor

- Aliases:
- Used in the guide:
- __init__

Class `SparseTensor`

Represents a sparse tensor.

Aliases:

- Class `tf.SparseTensor`
- Class `tf.compat.v1.SparseTensor`
- Class `tf.compat.v1.sparse.SparseTensor`
- Class `tf.compat.v2.SparseTensor`
- Class `tf.compat.v2.sparse.SparseTensor`
- Class `tf.sparse.SparseTensor`

Defined in `python/framework/sparse_tensor.py`.

Used in the guide:

- Ragged Tensors

TensorFlow represents a sparse tensor as three separate dense tensors: `indices`, `values`, and `dense_shape`. In Python, the three tensors are collected into a `SparseTensor` class for ease of use. If you have separate `indices`, `values`, and `dense_shape` tensors, wrap them in a `SparseTensor`object before passing to the ops below.

Concretely, the sparse tensor `SparseTensor(indices, values, dense_shape)` comprises the following components, where `N` and `ndims` are the number of values and number of dimensions in the `SparseTensor`, respectively:

- `indices`: A 2-D int64 tensor of dense_shape `[N, ndims]`, which specifies the indices of the elements in the sparse tensor that contain nonzero values (elements are zero-indexed). For example, `indices=[[1,3], [2,4]]` specifies that the elements with indexes of [1,3] and [2,4] have nonzero values.
- `values`: A 1-D tensor of any type and dense_shape `[N]`, which supplies the values for each element in `indices`. For example, given `indices=[[1,3], [2,4]]`, the parameter `values=[18, 3.6]` specifies that element [1,3] of the sparse tensor has a value of 18, and element [2,4] of the tensor has a value of 3.6.
- `dense_shape`: A 1-D int64 tensor of dense_shape `[ndims]`, which specifies the dense_shape of the sparse tensor. Takes a list indicating the number of elements in each dimension. For example, `dense_shape=[3,6]` specifies a two-dimensional 3x6 tensor, `dense_shape=[2,3,4]`specifies a three-dimensional 2x3x4 tensor, and `dense_shape=[9]` specifies a one-dimensional tensor with 9 elements.

The corresponding dense tensor satisfies:

```
dense.shape = dense_shape
dense[tuple(indices[i])] = values[i]
```

By convention, `indices` should be sorted in row-major order (or equivalently lexicographic order on the tuples `indices[i]`). This is not enforced when `SparseTensor` objects are constructed, but most ops assume correct ordering. If the ordering of sparse tensor `st` is wrong, a fixed version can be obtained by calling `tf.sparse.reorder(st)`.

Example: The sparse tensor

```
SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4])
```

represents the dense tensor

```
[[1, 0, 0, 0]
 [0, 0, 2, 0]
 [0, 0, 0, 0]]
```

`__init__`

```
__init__(
    indices,
    values,
    dense_shape
)
```

Creates a `SparseTensor`.

Args:

- **`indices`**: A 2-D int64 tensor of shape `[N, ndims]`.
- **`values`**: A 1-D tensor of any type and shape `[N]`.
- **`dense_shape`**: A 1-D int64 tensor of shape `[ndims]`.

Properties

`dense_shape`

A 1-D Tensor of int64 representing the shape of the dense tensor.

`dtype`

The `DType` of elements in this tensor.

`graph`

The `Graph` that contains the index, value, and dense_shape tensors.

`indices`

The indices of non-zero values in the represented dense tensor.

Returns:

A 2-D Tensor of int64 with dense_shape `[N, ndims]`, where `N` is the number of non-zero values in the tensor, and `ndims` is the rank.

`op`

The `Operation` that produces `values` as an output.

`shape`

Get the `TensorShape` representing the shape of the dense tensor.

Returns:

A `TensorShape` object.

`values`

The non-zero values in the represented dense tensor.

Returns:

A 1-D Tensor of any data type.

Methods

`__div__`

```
__div__(
    sp_x,
```

```
    y
)
```

Component-wise divides a SparseTensor by a dense Tensor.
Limitation: this Op only broadcasts the dense side to the sparse side, but not the other direction.

Args:

- **`sp_indices`**: A `Tensor` of type `int64`. 2-D. `N x R` matrix with the indices of non-empty values in a SparseTensor, possibly not in canonical ordering.
- **`sp_values`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`. 1-D. `N` non-empty values corresponding to `sp_indices`.
- **`sp_shape`**: A `Tensor` of type `int64`. 1-D. Shape of the input SparseTensor.
- **`dense`**: A `Tensor`. Must have the same type as `sp_values`. `R`-D. The dense Tensor operand.
- **`name`**: A name for the operation (optional).

Returns:

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

`__mul__`

```
__mul__(
    sp_x,
    y
)
```

Component-wise multiplies a SparseTensor by a dense Tensor.
The output locations corresponding to the implicitly zero elements in the sparse tensor will be zero (i.e., will not take up storage space), regardless of the contents of the dense tensor (even if it's +/-INF and that INF*0 == NaN).
Limitation: this Op only broadcasts the dense side to the sparse side, but not the other direction.

Args:

- **`sp_indices`**: A `Tensor` of type `int64`. 2-D. `N x R` matrix with the indices of non-empty values in a SparseTensor, possibly not in canonical ordering.
- **`sp_values`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`. 1-D. `N` non-empty values corresponding to `sp_indices`.
- **`sp_shape`**: A `Tensor` of type `int64`. 1-D. Shape of the input SparseTensor.
- **`dense`**: A `Tensor`. Must have the same type as `sp_values`. `R`-D. The dense Tensor operand.
- **`name`**: A name for the operation (optional).

Returns:

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

`__truediv__`

```
__truediv__(
    sp_x,
    y
)
```

Internal helper function for 'sp_t / dense_t'.

consumers

```
consumers()
```

eval

```
eval(
    feed_dict=None,
    session=None
)
```

Evaluates this sparse tensor in a `Session`.
Calling this method will execute all preceding operations that produce the inputs needed for the operation that produces this tensor.
N.B. Before invoking `SparseTensor.eval()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

Args:

- **`feed_dict`**: A dictionary that maps `Tensor` objects to feed values. See `tf.Session.run` for a description of the valid feed values.
- **`session`**: (Optional.) The `Session` to be used to evaluate this sparse tensor. If none, the default session will be used.

Returns:
A `SparseTensorValue` object.

from_value

```
@classmethod
from_value(
    cls,
    sparse_tensor_value
)
```

get_shape

```
get_shape()
```

Get the `TensorShape` representing the shape of the dense tensor.

Returns:
A `TensorShape` object.

tf.sparse.sparse_dense_matmul

- Contents
- Aliases:

Multiply SparseTensor (of rank 2) "A" by dense matrix "B".

Aliases:

- `tf.compat.v1.sparse.matmul`
- `tf.compat.v1.sparse.sparse_dense_matmul`
- `tf.compat.v1.sparse_tensor_dense_matmul`
- `tf.compat.v2.sparse.sparse_dense_matmul`
- `tf.sparse.sparse_dense_matmul`

```
tf.sparse.sparse_dense_matmul(
    sp_a,
    b,
    adjoint_a=False,
    adjoint_b=False,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.
No validity checking is performed on the indices of `A`. However, the following input format is recommended for optimal behavior:

- If `adjoint_a == false`: `A` should be sorted in lexicographically increasing order. Use `sparse.reorder` if you're not sure.
- If `adjoint_a == true`: `A` should be sorted in order of increasing dimension 1 (i.e., "column major" order instead of "row major" order).

Using `tf.nn.embedding_lookup_sparse` for sparse multiplication:
It's not obvious but you can consider `embedding_lookup_sparse` as another sparse and dense multiplication. In some situations, you may prefer to use `embedding_lookup_sparse` even though you're not dealing with embeddings.
There are two questions to ask in the decision process: Do you need gradients computed as sparse too? Is your sparse data represented as two `SparseTensor`s: ids and values? There is more explanation about data format below. If you answer any of these questions as yes, consider using`tf.nn.embedding_lookup_sparse`.
Following explains differences between the expected SparseTensors: For example if dense form of your sparse data has shape `[3, 5]` and values:

```
[[  a      ]
 [b       c]
 [    d    ]]
```

`SparseTensor` format expected by `sparse_tensor_dense_matmul`: `sp_a` (indices, values):

```
[0, 1]: a
[1, 0]: b
[1, 4]: c
[2, 2]: d
```

`SparseTensor` format expected by `embedding_lookup_sparse`: `sp_ids` `sp_weights`

```
[0, 0]: 1                [0, 0]: a
[1, 0]: 0                [1, 0]: b
[1, 1]: 4                [1, 1]: c
[2, 0]: 2                [2, 0]: d
```

Deciding when to use `sparse_tensor_dense_matmul` vs. `matmul`(a_is_sparse=True):
There are a number of questions to ask in the decision process, including:

- Will the SparseTensor `A` fit in memory if densified?
- Is the column count of the product large (>> 1)?
- Is the density of `A` larger than approximately 15%?

If the answer to several of these questions is yes, consider converting the `SparseTensor` to a dense one and using `tf.matmul` with `a_is_sparse=True`.
This operation tends to perform well when `A` is more sparse, if the column size of the product is small (e.g. matrix-vector multiplication), if `sp_a.dense_shape` takes on large values.

Below is a rough speed comparison between `sparse_tensor_dense_matmul`, labeled 'sparse', and `matmul`(a_is_sparse=True), labeled 'dense'. For purposes of the comparison, the time spent converting from a `SparseTensor` to a dense `Tensor` is not included, so it is overly conservative with respect to the time ratio.

Benchmark system:
CPU: Intel Ivybridge with HyperThreading (6 cores) dL1:32KB dL2:256KB dL3:12MB GPU: NVidia Tesla k40c

Compiled with:
`-c opt --config=cuda --copt=-mavx`

```
tensorflow/python/sparse_tensor_dense_matmul_op_test --benchmarks
A sparse [m, k] with % nonzero values between 1% and 80%
B dense [k, n]

% nnz  n   gpu   m     k     dt(dense)     dt(sparse)   dt(sparse)/dt(dense)
0.01   1   True  100   100   0.000221166   0.00010154   0.459112
0.01   1   True  100   1000  0.00033858    0.000109275  0.322745
0.01   1   True  1000  100   0.000310557   9.85661e-05  0.317385
0.01   1   True  1000  1000  0.0008721     0.000100875  0.115669
0.01   1   False 100   100   0.000208085   0.000107603  0.51711
0.01   1   False 100   1000  0.000327112   9.51118e-05  0.290762
0.01   1   False 1000  100   0.000308222   0.00010345   0.335635
0.01   1   False 1000  1000  0.000865721   0.000101397  0.117124
0.01   10  True  100   100   0.000218522   0.000105537  0.482958
0.01   10  True  100   1000  0.000340882   0.000111641  0.327506
0.01   10  True  1000  100   0.000315472   0.000117376  0.372064
0.01   10  True  1000  1000  0.000905493   0.000123263  0.136128
0.01   10  False 100   100   0.000221529   9.82571e-05  0.44354
0.01   10  False 100   1000  0.000330552   0.000112615  0.340687
0.01   10  False 1000  100   0.000341277   0.000114097  0.334324
0.01   10  False 1000  1000  0.000819944   0.000120982  0.147549
0.01   25  True  100   100   0.000207806   0.000105977  0.509981
0.01   25  True  100   1000  0.000322879   0.00012921   0.400181
0.01   25  True  1000  100   0.00038262    0.00014158   0.370035
0.01   25  True  1000  1000  0.000865438   0.000202083  0.233504
0.01   25  False 100   100   0.000209401   0.000104696  0.499979
0.01   25  False 100   1000  0.000321161   0.000130737  0.407076
0.01   25  False 1000  100   0.000377012   0.000136801  0.362856
0.01   25  False 1000  1000  0.000861125   0.00020272   0.235413
0.2    1   True  100   100   0.000206952   9.69219e-05  0.46833
0.2    1   True  100   1000  0.000348674   0.000147475  0.422959
0.2    1   True  1000  100   0.000336908   0.00010122   0.300439
0.2    1   True  1000  1000  0.001022      0.000203274  0.198898
0.2    1   False 100   100   0.000207532   9.5412e-05   0.459746
0.2    1   False 100   1000  0.000356127   0.000146824  0.41228
0.2    1   False 1000  100   0.000322664   0.000100918  0.312764
0.2    1   False 1000  1000  0.000998987   0.000203442  0.203648
0.2    10  True  100   100   0.000211692   0.000109903  0.519165
0.2    10  True  100   1000  0.000372819   0.000164321  0.440753
```

0.2	10	True	1000	100	0.000338651	0.000144806	0.427596
0.2	10	True	1000	1000	0.00108312	0.000758876	0.70064
0.2	10	False	100	100	0.000215727	0.000110502	0.512231
0.2	10	False	100	1000	0.000375419	0.0001613	0.429653
0.2	10	False	1000	100	0.000336999	0.000145628	0.432132
0.2	10	False	1000	1000	0.00110502	0.000762043	0.689618
0.2	25	True	100	100	0.000218705	0.000129913	0.594009
0.2	25	True	100	1000	0.000394794	0.00029428	0.745402
0.2	25	True	1000	100	0.000404483	0.0002693	0.665788
0.2	25	True	1000	1000	0.0012002	0.00194494	1.62052
0.2	25	False	100	100	0.000221494	0.0001306	0.589632
0.2	25	False	100	1000	0.000396436	0.000297204	0.74969
0.2	25	False	1000	100	0.000409346	0.000270068	0.659754
0.2	25	False	1000	1000	0.00121051	0.00193737	1.60046
0.5	1	True	100	100	0.000214981	9.82111e-05	0.456836
0.5	1	True	100	1000	0.000415328	0.000223073	0.537101
0.5	1	True	1000	100	0.000358324	0.00011269	0.314492
0.5	1	True	1000	1000	0.00137612	0.000437401	0.317851
0.5	1	False	100	100	0.000224196	0.000101423	0.452386
0.5	1	False	100	1000	0.000400987	0.000223286	0.556841
0.5	1	False	1000	100	0.000368825	0.00011224	0.304318
0.5	1	False	1000	1000	0.00136036	0.000429369	0.31563
0.5	10	True	100	100	0.000222125	0.000112308	0.505608
0.5	10	True	100	1000	0.000461088	0.00032357	0.701753
0.5	10	True	1000	100	0.000394624	0.000225497	0.571422
0.5	10	True	1000	1000	0.00158027	0.00190898	1.20801
0.5	10	False	100	100	0.000232083	0.000114978	0.495418
0.5	10	False	100	1000	0.000454574	0.000324632	0.714146
0.5	10	False	1000	100	0.000379097	0.000227768	0.600817
0.5	10	False	1000	1000	0.00160292	0.00190168	1.18638
0.5	25	True	100	100	0.00023429	0.000151703	0.647501
0.5	25	True	100	1000	0.000497462	0.000598873	1.20386
0.5	25	True	1000	100	0.000460778	0.000557038	1.20891
0.5	25	True	1000	1000	0.00170036	0.00467336	2.74845
0.5	25	False	100	100	0.000228981	0.000155334	0.678371
0.5	25	False	100	1000	0.000496139	0.000620789	1.25124
0.5	25	False	1000	100	0.00045473	0.000551528	1.21287
0.5	25	False	1000	1000	0.00171793	0.00467152	2.71927
0.8	1	True	100	100	0.000222037	0.000105301	0.47425
0.8	1	True	100	1000	0.000410804	0.000329327	0.801664
0.8	1	True	1000	100	0.000349735	0.000131225	0.375212
0.8	1	True	1000	1000	0.00139219	0.000677065	0.48633
0.8	1	False	100	100	0.000214079	0.000107486	0.502085
0.8	1	False	100	1000	0.000413746	0.000323244	0.781261
0.8	1	False	1000	100	0.000348983	0.000131983	0.378193
0.8	1	False	1000	1000	0.00136296	0.000685325	0.50282
0.8	10	True	100	100	0.000229159	0.00011825	0.516017
0.8	10	True	100	1000	0.000498845	0.000532618	1.0677

```
0.8     10  True   1000  100   0.000383126   0.00029935   0.781336
0.8     10  True   1000  1000  0.00162866    0.00307312   1.88689
0.8     10  False  100   100   0.000230783   0.000124958  0.541452
0.8     10  False  100   1000  0.000493393   0.000550654  1.11606
0.8     10  False  1000  100   0.000377167   0.000298581  0.791642
0.8     10  False  1000  1000  0.00165795    0.00305103   1.84024
0.8     25  True   100   100   0.000233496   0.000175241  0.75051
0.8     25  True   100   1000  0.00055654    0.00102658   1.84458
0.8     25  True   1000  100   0.000463814   0.000783267  1.68875
0.8     25  True   1000  1000  0.00186905    0.00755344   4.04132
0.8     25  False  100   100   0.000240243   0.000175047  0.728625
0.8     25  False  100   1000  0.000578102   0.00104499   1.80763
0.8     25  False  1000  100   0.000485113   0.000776849  1.60138
0.8     25  False  1000  1000  0.00211448    0.00752736   3.55992
```

Args:

- **`sp_a`**: SparseTensor A, of rank 2.
- **`b`**: A dense Matrix with the same dtype as sp_a.
- **`adjoint_a`**: Use the adjoint of A in the matrix multiply. If A is complex, this is transpose(conj(A)). Otherwise it's transpose(A).
- **`adjoint_b`**: Use the adjoint of B in the matrix multiply. If B is complex, this is transpose(conj(B)). Otherwise it's transpose(B).
- **`name`**: A name prefix for the returned tensors (optional)

Returns:

A dense matrix (pseudo-code in dense np.matrix notation): `A = A.H if adjoint_a else A` `B = B.H if adjoint_b else B` `return A*B`

tf.sparse.split

- Contents
- Aliases:

Split a `SparseTensor` into `num_split` tensors along `axis`.

Aliases:

- `tf.compat.v2.sparse.split`
- `tf.sparse.split`

```
tf.sparse.split(
    sp_input=None,
    num_split=None,
    axis=None,
    name=None
)

```

Defined in `python/ops/sparse_ops.py`.

If the `sp_input.dense_shape[axis]` is not an integer multiple of `num_split` each slice starting from 0:`shape[axis] % num_split` gets extra one dimension. For example, if `axis = 1` and `num_split = 2` and the input is:

```
input_tensor = shape = [2, 7]
[    a   d e  ]
```

```
[b c          ]
```

Graphically the output tensors are:

```
output_tensor[0] =
[    a ]
[b c   ]

output_tensor[1] =
[ d e  ]
[      ]
```

Args:

- **`sp_input`**: The `SparseTensor` to split.
- **`num_split`**: A Python integer. The number of ways to split.
- **`axis`**: A 0-D `int32` `Tensor`. The dimension along which to split.
- **`name`**: A name for the operation (optional).

Returns:

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

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.to_dense

- Contents
- Aliases:
- Used in the guide:

Converts a `SparseTensor` into a dense tensor.

Aliases:

- `tf.compat.v1.sparse.to_dense`
- `tf.compat.v1.sparse_tensor_to_dense`
- `tf.compat.v2.sparse.to_dense`
- `tf.sparse.to_dense`

```
tf.sparse.to_dense(
    sp_input,
    default_value=0,
    validate_indices=True,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

Used in the guide:

- Ragged Tensors

This op is a convenience wrapper around `sparse_to_dense` for `SparseTensor`s.

For example, if `sp_input` has shape `[3, 5]` and non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
```

and `default_value` is `x`, then the output will be a dense `[3, 5]` string tensor with values:

```
[[x a x b x]
 [x x x x x]
 [c x x x x]]
```

Indices must be without repeats. This is only tested if `validate_indices` is `True`.

Args:

- **`sp_input`**: The input `SparseTensor`.
- **`default_value`**: Scalar value to set for indices not specified in `sp_input`. Defaults to zero.
- **`validate_indices`**: A boolean value. If `True`, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats.
- **`name`**: A name prefix for the returned tensors (optional).

Returns:

A dense tensor with shape `sp_input.dense_shape` and values specified by the non-empty values in `sp_input`. Indices not in `sp_input` are assigned `default_value`.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.to_indicator

- Contents
- Aliases:

Converts a `SparseTensor` of ids into a dense bool indicator tensor.

Aliases:

- `tf.compat.v1.sparse.to_indicator`
- `tf.compat.v1.sparse_to_indicator`
- `tf.compat.v2.sparse.to_indicator`
- `tf.sparse.to_indicator`

```
tf.sparse.to_indicator(
    sp_input,
    vocab_size,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

The last dimension of `sp_input.indices` is discarded and replaced with the values of `sp_input`. If `sp_input.dense_shape = [D0, D1, ..., Dn, K]`, then `output.shape = [D0, D1, ..., Dn, vocab_size]`, where

```
output[d_0, d_1, ..., d_n, sp_input[d_0, d_1, ..., d_n, k]] = True
```

and False elsewhere in `output`.

For example, if `sp_input.dense_shape = [2, 3, 4]` with non-empty values:

```
[0, 0, 0]: 0
[0, 1, 0]: 10
[1, 0, 3]: 103
[1, 1, 2]: 150
[1, 1, 3]: 149
[1, 1, 4]: 150
```

```
[1, 2, 1]: 121
```

and `vocab_size = 200`, then the output will be a `[2, 3, 200]` dense bool tensor with False everywhere except at positions

```
(0, 0, 0), (0, 1, 10), (1, 0, 103), (1, 1, 149), (1, 1, 150),
(1, 2, 121).
```

Note that repeats are allowed in the input SparseTensor. This op is useful for converting `SparseTensor`s into dense formats for compatibility with ops that expect dense tensors. The input `SparseTensor` must be in row-major order.

Args:

- **`sp_input`**: A `SparseTensor` with `values` property of type `int32` or `int64`.
- **`vocab_size`**: A scalar int64 Tensor (or Python int) containing the new size of the last dimension, `all(0 <= sp_input.values < vocab_size)`.
- **`name`**: A name prefix for the returned tensors (optional)

Returns:

A dense bool indicator tensor representing the indices with specified value.

Raises:

- **`TypeError`**: If `sp_input` is not a `SparseTensor`.

tf.sparse.transpose

- Contents
- Aliases:

Transposes a `SparseTensor`

Aliases:

- `tf.compat.v1.sparse.transpose`
- `tf.compat.v1.sparse_transpose`
- `tf.compat.v2.sparse.transpose`
- `tf.sparse.transpose`

```
tf.sparse.transpose(
    sp_input,
    perm=None,
    name=None
)
```

Defined in `python/ops/sparse_ops.py`.

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, if `sp_input` has shape `[4, 5]` and `indices` / `values`:

```
[0, 3]: b
[0, 1]: a
[3, 1]: d
[2, 0]: c
```

then the output will be a `SparseTensor` of shape `[5, 4]` and `indices` / `values`:

```
[0, 2]: c
[1, 0]: a
```

```
[1, 3]: d
[3, 0]: b
```

Args:
- `sp_input`: The input `SparseTensor`.
- `perm`: A permutation of the dimensions of `sp_input`.
- `name`: A name prefix for the returned tensors (optional)

Returns:
A transposed `SparseTensor`.

Raises:
- `TypeError`: If `sp_input` is not a `SparseTensor`.

Module: tf.compat.v1.strings / tf.strings

- Contents
- Functions

Operations for working with string Tensors.

Functions

`as_string(...)`: Converts each entry in the given tensor to strings. Supports many numeric
`bytes_split(...)`: Split string elements of `input` into bytes.
`format(...)`: Formats a string template using a list of tensors.
`join(...)`: Joins the strings in the given list of string tensors into one tensor;
`length(...)`: String lengths of `input`.
`lower(...)`: TODO: add doc.
`reduce_join(...)`: Joins a string Tensor across the given dimensions.
`regex_full_match(...)`: Check if the input matches the regex pattern.
`regex_replace(...)`: Replace elements of `input` matching regex `pattern` with `rewrite`.
`split(...)`: Split elements of `input` based on `sep`.
`strip(...)`: Strip leading and trailing whitespaces from the Tensor.
`substr(...)`: Return substrings from `Tensor` of strings.
`to_hash_bucket(...)`: Converts each string in the input Tensor to its hash mod by a number of buckets.
`to_hash_bucket_fast(...)`: Converts each string in the input Tensor to its hash mod by a number of buckets.
`to_hash_bucket_strong(...)`: Converts each string in the input Tensor to its hash mod by a number of buckets.
`to_number(...)`: Converts each string in the input Tensor to the specified numeric type.
`unicode_decode(...)`: Decodes each string in `input` into a sequence of Unicode code points.
`unicode_decode_with_offsets(...)`: Decodes each string into a sequence of code points with start offsets.
`unicode_encode(...)`: Encodes each sequence of Unicode code points in `input` into a string.
`unicode_script(...)`: Determine the script codes of a given tensor of Unicode integer code points.
`unicode_split(...)`: Splits each string in `input` into a sequence of Unicode code points.
`unicode_split_with_offsets(...)`: Splits each string into a sequence of code points with start offsets.
`unicode_transcode(...)`: Transcode the input text from a source encoding to a destination encoding.
`upper(...)`: TODO: add doc.

tf.compat.v1.strings.length

String lengths of `input`.

```
tf.compat.v1.strings.length(
    input,
    name=None,
    unit='BYTE'
)
```

Defined in [python/ops/string_ops.py]().

Computes the length of each string given in the input tensor.

Args:

- **`input`**: A `Tensor` of type `string`. The string for which to compute the length.
- **`unit`**: An optional `string` from: `"BYTE", "UTF8_CHAR"`. Defaults to `"BYTE"`. The unit that is counted to compute string length. One of: `"BYTE"` (for the number of bytes in each string) or `"UTF8_CHAR"` (for the number of UTF-8 encoded Unicode code points in each string). Results are undefined if `unit=UTF8_CHAR` and the `input` strings do not contain structurally valid UTF-8.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

tf.compat.v1.strings.split

Split elements of `input` based on `sep`.

```
tf.compat.v1.strings.split(
    input=None,
    sep=None,
    maxsplit=-1,
    result_type='SparseTensor',
    source=None,
    name=None
)
```

Defined in [python/ops/ragged/ragged_string_ops.py]().

Let N be the size of `input` (typically N will be the batch size). Split each element of `input` based on `sep` and return a `SparseTensor` or `RaggedTensor` containing the split tokens. Empty tokens are ignored.

Examples:

```
>>> tf.strings.split(['hello world', 'a b c'])
tf.SparseTensor(indices=[[0, 0], [0, 1], [1, 0], [1, 1], [1, 2]],
                values=['hello', 'world', 'a', 'b', 'c']
                dense_shape=[2, 3])

>>> tf.strings.split(['hello world', 'a b c'], result_type="RaggedTensor")
<tf.RaggedTensor [['hello', 'world'], ['a', 'b', 'c']]>
```

If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings. For example, `input` of `"1<>2<><>3"` and `sep` of `"<>"` returns `["1", "2", "", "3"]`.

If `sep` is None or an empty string, consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace.

Note that the above mentioned behavior matches python's str.split.

Args:

- **`input`**: A string `Tensor` of rank `N`, the strings to split. If `rank(input)` is not known statically, then it is assumed to be `1`.
- **`sep`**: `0-D` string `Tensor`, the delimiter character.
- **`maxsplit`**: An `int`. If `maxsplit > 0`, limit of the split of the result.
- **`result_type`**: The tensor type for the result: one of `"RaggedTensor"` or `"SparseTensor"`.
- **`source`**: alias for "input" argument.
- **`name`**: A name for the operation (optional).

Raises:

- **`ValueError`**: If sep is not a string.

Returns:

A `SparseTensor` or `RaggedTensor` of rank `N+1`, the strings split according to the delimiter.

tf.compat.v1.strings.substr

Return substrings from `Tensor` of strings.

```
tf.compat.v1.strings.substr(
    input,
    pos,
    len,
    name=None,
    unit='BYTE'
)
```

Defined in `python/ops/string_ops.py`.

For each string in the input `Tensor`, creates a substring starting at index `pos` with a total length of `len`.

If `len` defines a substring that would extend beyond the length of the input string, then as many characters as possible are used.

A negative `pos` indicates distance within the string backwards from the end.

If `pos` specifies an index which is out of range for any of the input strings, then an `InvalidArgumentError` is thrown.

`pos` and `len` must have the same shape, otherwise a `ValueError` is thrown on Op creation.

NOTE: `Substr` supports broadcasting up to two dimensions. More about broadcasting here

---

Examples

Using scalar `pos` and `len`:

```
input = [b'Hello', b'World']
position = 1
length = 3

output = [b'ell', b'orl']
```

Using `pos` and `len` with same shape as `input`:

```
input = [[b'ten', b'eleven', b'twelve'],
         [b'thirteen', b'fourteen', b'fifteen'],
         [b'sixteen', b'seventeen', b'eighteen']]
position = [[1, 2, 3],
            [1, 2, 3],
            [1, 2, 3]]
```

```
length =   [[2, 3, 4],
            [4, 3, 2],
            [5, 5, 5]]

output = [[b'en', b'eve', b'lve'],
          [b'hirt', b'urt', b'te'],
          [b'ixtee', b'vente', b'hteen']]
```

Broadcasting `pos` and `len` onto `input`:

```
input = [[b'ten', b'eleven', b'twelve'],
         [b'thirteen', b'fourteen', b'fifteen'],
         [b'sixteen', b'seventeen', b'eighteen'],
         [b'nineteen', b'twenty', b'twentyone']]
position = [1, 2, 3]
length =   [1, 2, 3]

output = [[b'e', b'ev', b'lve'],
          [b'h', b'ur', b'tee'],
          [b'i', b've', b'hte'],
          [b'i', b'en', b'nty']]
```

Broadcasting `input` onto `pos` and `len`:

```
input = b'thirteen'
position = [1, 5, 7]
length =   [3, 2, 1]

output = [b'hir', b'ee', b'n']
```

Args:

- **`input`**: A `Tensor` of type `string`. Tensor of strings
- **`pos`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. Scalar defining the position of first character in each substring
- **`len`**: A `Tensor`. Must have the same type as `pos`. Scalar defining the number of characters to include in each substring
- **`unit`**: An optional `string` from: `"BYTE", "UTF8_CHAR"`. Defaults to `"BYTE"`. The unit that is used to create the substring. One of: `"BYTE"` (for defining position and length by bytes) or `"UTF8_CHAR"` (for the UTF-8 encoded Unicode code points). The default is `"BYTE"`. Results are undefined if `unit=UTF8_CHAR` and the `input` strings do not contain structurally valid UTF-8.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.strings.as_string

- Contents
- Aliases:

Converts each entry in the given tensor to strings. Supports many numeric

Aliases:

- `tf.as_string`

- `tf.compat.v1.as_string`
- `tf.compat.v1.dtypes.as_string`
- `tf.compat.v1.strings.as_string`
- `tf.compat.v2.as_string`
- `tf.compat.v2.strings.as_string`
- `tf.strings.as_string`

```
tf.strings.as_string(
    input,
    precision=-1,
    scientific=False,
    shortest=False,
    width=-1,
    fill='',
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.
types and boolean.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `float32`, `float64`, `bool`.
- **`precision`**: An optional `int`. Defaults to `-1`. The post-decimal precision to use for floating point numbers. Only used if precision > -1.
- **`scientific`**: An optional `bool`. Defaults to `False`. Use scientific notation for floating point numbers.
- **`shortest`**: An optional `bool`. Defaults to `False`. Use shortest representation (either scientific or standard) for floating point numbers.
- **`width`**: An optional `int`. Defaults to `-1`. Pad pre-decimal numbers to this width. Applies to both floating point and integer numbers. Only used if width > -1.
- **`fill`**: An optional `string`. Defaults to `""`. The value to pad if width > -1. If empty, pads with spaces. Another typical value is '0'. String cannot be longer than 1 character.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.strings.bytes_split

- Contents
- Aliases:

Split string elements of `input` into bytes.

Aliases:

- `tf.compat.v1.strings.bytes_split`
- `tf.compat.v2.strings.bytes_split`
- `tf.strings.bytes_split`

```
tf.strings.bytes_split(
    input,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Examples:

```
>>> tf.strings.to_bytes('hello')
['h', 'e', 'l', 'l', 'o']
>>> tf.strings.to_bytes(['hello', '123'])
<RaggedTensor [['h', 'e', 'l', 'l', 'o'], ['1', '2', '3']]>
```

Note that this op splits strings into bytes, not unicode characters. To split strings into unicode characters, use `tf.strings.unicode_split`.
See also: `tf.io.decode_raw`, `tf.strings.split`, `tf.strings.unicode_split`.

Args:

- **`input`**: A string `Tensor` or `RaggedTensor`: the strings to split. Must have a statically known rank (`N`).
- **`name`**: A name for the operation (optional).

Returns:

A `RaggedTensor` of rank `N+1`: the bytes that make up the soruce strings.

tf.strings.format

- Contents
- Aliases:

Formats a string template using a list of tensors.

Aliases:

- `tf.compat.v1.strings.format`
- `tf.compat.v2.strings.format`
- `tf.strings.format`

```
tf.strings.format(
    template,
    inputs,
    placeholder='{}',
    summarize=3,
    name=None
)
```

Defined in `python/ops/string_ops.py`.
Formats a string template using a list of tensors, abbreviating tensors by only printing the first and last `summarize` elements of each dimension (recursively). If formatting only one tensor into a template, the tensor does not have to be wrapped in a list.

Example:

Formatting a single-tensor template:

```
sess = tf.compat.v1.Session()
with sess.as_default():
    tensor = tf.range(10)
    formatted = tf.strings.format("tensor: {}, suffix", tensor)
    out = sess.run(formatted)
    expected = "tensor: [0 1 2 ... 7 8 9], suffix"

    assert(out.decode() == expected)
```

Formatting a multi-tensor template:

```
sess = tf.compat.v1.Session()
with sess.as_default():
    tensor_one = tf.reshape(tf.range(100), [10, 10])
    tensor_two = tf.range(10)
    formatted = tf.strings.format("first: {}, second: {}, suffix",
      (tensor_one, tensor_two))

    out = sess.run(formatted)
    expected = ("first: [[0 1 2 ... 7 8 9]\n"
          " [10 11 12 ... 17 18 19]\n"
          " [20 21 22 ... 27 28 29]\n"
          " ...\n"
          " [70 71 72 ... 77 78 79]\n"
          " [80 81 82 ... 87 88 89]\n"
          " [90 91 92 ... 97 98 99]], second: [0 1 2 ... 7 8 9], suffix")

    assert(out.decode() == expected)
```

Args:

- **`template`**: A string template to format tensor values into.
- **`inputs`**: A list of `Tensor` objects, or a single Tensor. The list of tensors to format into the template string. If a solitary tensor is passed in, the input tensor will automatically be wrapped as a list.
- **`placeholder`**: An optional `string`. Defaults to `{}`. At each placeholder occurring in the template, a subsequent tensor will be inserted.
- **`summarize`**: An optional `int`. Defaults to `3`. When formatting the tensors, show the first and last `summarize` entries of each tensor dimension (recursively). If set to -1, all elements of the tensor will be shown.
- **`name`**: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`.

Raises:

- **`ValueError`**: if the number of placeholders does not match the number of inputs.

tf.strings.join

- Contents
- Aliases:
- Used in the guide:

Joins the strings in the given list of string tensors into one tensor;

Aliases:

- `tf.compat.v1.string_join`
- `tf.compat.v1.strings.join`
- `tf.compat.v2.strings.join`
- `tf.strings.join`

```
tf.strings.join(
    inputs,
    separator='',
    name=None
```

```
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Used in the guide:

- Ragged Tensors

with the given separator (default is an empty separator).

Args:

- **`inputs`**: A list of at least 1 `Tensor` objects with type `string`. A list of string tensors. The tensors must all have the same shape, or be scalars. Scalars may be mixed in; these will be broadcast to the shape of non-scalar inputs.
- **`separator`**: An optional `string`. Defaults to `""`. string, an optional join separator.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.strings.length

- Contents
- Aliases:
- Used in the tutorials:

Aliases:

- `tf.compat.v2.strings.length`
- `tf.strings.length`

```
tf.strings.length(
    input,
    unit='BYTE',
    name=None
)
```

Defined in `python/ops/string_ops.py`.

Used in the tutorials:

- Unicode strings

tf.strings.lower

- Contents
- Aliases:

TODO: add doc.

Aliases:

- `tf.compat.v1.strings.lower`
- `tf.compat.v2.strings.lower`
- `tf.strings.lower`

```
tf.strings.lower(
    input,
    encoding='',
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Args:

- **`input`**: A `Tensor` of type `string`.
- **`encoding`**: An optional `string`. Defaults to `""`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.strings.reduce_join

- Contents
- Aliases:

Aliases:

- `tf.compat.v2.strings.reduce_join`
- `tf.strings.reduce_join`

```
tf.strings.reduce_join(
    inputs,
    axis=None,
    keepdims=False,
    separator='',
    name=None
)
```

Defined in `python/ops/string_ops.py`.

tf.strings.regex_full_match

- Contents
- Aliases:

Check if the input matches the regex pattern.

Aliases:

- `tf.compat.v1.strings.regex_full_match`
- `tf.compat.v2.strings.regex_full_match`
- `tf.strings.regex_full_match`

```
tf.strings.regex_full_match(
    input,
    pattern,
    name=None
)
```

Defined in `python/ops/string_ops.py`.

The input is a string tensor of any shape. The pattern is a scalar string tensor which is applied to every element of the input tensor. The boolean values (True or False) of the output tensor indicate if the input matches the regex pattern provided.

The pattern follows the re2 syntax (https://github.com/google/re2/wiki/Syntax)

Args:

- **`input`**: A `Tensor` of type `string`. A string tensor of the text to be processed.
- **`pattern`**: A `Tensor` of type `string`. A scalar string tensor containing the regular expression to match the input.

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

Returns:
A `Tensor` of type `bool`.

tf.strings.regex_replace

- Contents
- Aliases:
- Used in the tutorials:

Replace elements of `input` matching regex `pattern` with `rewrite`.

Aliases:

- `tf.compat.v1.regex_replace`
- `tf.compat.v1.strings.regex_replace`
- `tf.compat.v2.strings.regex_replace`
- `tf.strings.regex_replace`

```
tf.strings.regex_replace(
    input,
    pattern,
    rewrite,
    replace_global=True,
    name=None
)
```

Defined in `python/ops/string_ops.py`.

Used in the tutorials:

- Load CSV with tf.data

Args:

- `input`: string `Tensor`, the source strings to process.
- `pattern`: string or scalar string `Tensor`, regular expression to use, see more details at https://github.com/google/re2/wiki/Syntax
- `rewrite`: string or scalar string `Tensor`, value to use in match replacement, supports backslash-escaped digits (\1 to \9) can be to insert text matching corresponding parenthesized group.
- `replace_global`: `bool`, if `True` replace all non-overlapping matches, else replace only the first match.
- `name`: A name for the operation (optional).

Returns:
string `Tensor` of the same shape as `input` with specified replacements.

tf.strings.split

- Contents
- Aliases:

Split elements of `input` based on `sep` into a `RaggedTensor`.

Aliases:

- `tf.compat.v2.strings.split`
- `tf.strings.split`

```
tf.strings.split(
    input,
    sep=None,
```

```
    maxsplit=-1,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Let N be the size of `input` (typically N will be the batch size). Split each element of `input` based on `sep` and return a `SparseTensor` or `RaggedTensor` containing the split tokens. Empty tokens are ignored.

Example:

```
>>> tf.strings.split('hello world')
<Tensor ['hello', 'world']>
>>> tf.strings.split(['hello world', 'a b c'])
<tf.RaggedTensor [['hello', 'world'], ['a', 'b', 'c']]>
```

If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings. For example, `input` of `"1<>2<><>3"` and `sep` of `"<>"` returns `["1", "2", "", "3"]`.

If `sep` is None or an empty string, consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace.

Note that the above mentioned behavior matches python's str.split.

Args:

- **`input`**: A string `Tensor` of rank `N`, the strings to split. If `rank(input)` is not known statically, then it is assumed to be `1`.
- **`sep`**: `0-D` string `Tensor`, the delimiter string.
- **`maxsplit`**: An `int`. If `maxsplit > 0`, limit of the split of the result.
- **`name`**: A name for the operation (optional).

Raises:

- **`ValueError`**: If sep is not a string.

Returns:

A `RaggedTensor` of rank `N+1`, the strings split according to the delimiter.

tf.strings.strip

- Contents
- Aliases:

Strip leading and trailing whitespaces from the Tensor.

Aliases:

- `tf.compat.v1.string_strip`
- `tf.compat.v1.strings.strip`
- `tf.compat.v2.strings.strip`
- `tf.strings.strip`

```
tf.strings.strip(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Args:

- **`input`**: A `Tensor` of type `string`. A string `Tensor` of any shape.

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

  Returns:
  A `Tensor` of type `string`.

tf.strings.substr

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Return substrings from `Tensor` of strings.

Aliases:

- `tf.compat.v2.strings.substr`
- `tf.strings.substr`

```
tf.strings.substr(
    input,
    pos,
    len,
    unit='BYTE',
    name=None
)
```

Defined in `python/ops/string_ops.py`.

Used in the guide:

- Ragged Tensors

Used in the tutorials:

- Unicode strings

For each string in the input `Tensor`, creates a substring starting at index `pos` with a total length of `len`.
If `len` defines a substring that would extend beyond the length of the input string, then as many characters as possible are used.
A negative `pos` indicates distance within the string backwards from the end.
If `pos` specifies an index which is out of range for any of the input strings, then an `InvalidArgumentError` is thrown.
`pos` and `len` must have the same shape, otherwise a `ValueError` is thrown on Op creation.
NOTE: `Substr` supports broadcasting up to two dimensions. More about broadcasting here

---

Examples
Using scalar `pos` and `len`:

```
input = [b'Hello', b'World']
position = 1
length = 3

output = [b'ell', b'orl']
```

Using `pos` and `len` with same shape as `input`:

```
input = [[b'ten', b'eleven', b'twelve'],
         [b'thirteen', b'fourteen', b'fifteen'],
```

```
         [b'sixteen', b'seventeen', b'eighteen']]
position = [[1, 2, 3],
            [1, 2, 3],
            [1, 2, 3]]
length =   [[2, 3, 4],
            [4, 3, 2],
            [5, 5, 5]]

output = [[b'en', b'eve', b'lve'],
          [b'hirt', b'urt', b'te'],
          [b'ixtee', b'vente', b'hteen']]
```

Broadcasting `pos` and `len` onto `input`:

```
input = [[b'ten', b'eleven', b'twelve'],
         [b'thirteen', b'fourteen', b'fifteen'],
         [b'sixteen', b'seventeen', b'eighteen'],
         [b'nineteen', b'twenty', b'twentyone']]
position = [1, 2, 3]
length =   [1, 2, 3]

output = [[b'e', b'ev', b'lve'],
          [b'h', b'ur', b'tee'],
          [b'i', b've', b'hte'],
          [b'i', b'en', b'nty']]
```

Broadcasting `input` onto `pos` and `len`:

```
input = b'thirteen'
position = [1, 5, 7]
length =   [3, 2, 1]

output = [b'hir', b'ee', b'n']
```

Args:

- **`input`**: A `Tensor` of type `string`. Tensor of strings
- **`pos`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. Scalar defining the position of first character in each substring
- **`len`**: A `Tensor`. Must have the same type as `pos`. Scalar defining the number of characters to include in each substring
- **`unit`**: An optional `string` from: `"BYTE", "UTF8_CHAR"`. Defaults to `"BYTE"`. The unit that is used to create the substring. One of: `"BYTE"` (for defining position and length by bytes) or `"UTF8_CHAR"` (for the UTF-8 encoded Unicode code points). The default is `"BYTE"`. Results are undefined if `unit=UTF8_CHAR` and the `input` strings do not contain structurally valid UTF-8.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

tf.strings.to_hash_bucket

- Contents

- Aliases:

Converts each string in the input Tensor to its hash mod by a number of buckets.

Aliases:

- `tf.compat.v2.strings.to_hash_bucket`
- `tf.strings.to_hash_bucket`

```
tf.strings.to_hash_bucket(
    input,
    num_buckets,
    name=None
)
```

Defined in `python/ops/string_ops.py`.
The hash function is deterministic on the content of the string within the process.
Note that the hash function may change from time to time. This functionality will be deprecated and it's recommended to
use `tf.strings.to_hash_bucket_fast()` or `tf.strings.to_hash_bucket_strong()`.

Args:

- **`input`**: A `Tensor` of type `string`.
- **`num_buckets`**: An `int` that is `>= 1`. The number of buckets.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `int64`.

tf.strings.to_hash_bucket_fast

- Contents
- Aliases:
- Used in the guide:

Converts each string in the input Tensor to its hash mod by a number of buckets.

Aliases:

- `tf.compat.v1.string_to_hash_bucket_fast`
- `tf.compat.v1.strings.to_hash_bucket_fast`
- `tf.compat.v2.strings.to_hash_bucket_fast`
- `tf.strings.to_hash_bucket_fast`

```
tf.strings.to_hash_bucket_fast(
    input,
    num_buckets,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Used in the guide:

- Ragged Tensors

The hash function is deterministic on the content of the string within the process and will never change. However, it is not suitable for cryptography. This function may be used when CPU time is scarce and inputs are trusted or unimportant. There is a risk of adversaries constructing inputs that all hash to the same bucket. To prevent this problem, use a strong hash function with`tf.string_to_hash_bucket_strong`.

Args:

- **`input`**: A `Tensor` of type `string`. The strings to assign a hash bucket.
- **`num_buckets`**: An `int` that is `>= 1`. The number of buckets.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

tf.strings.to_hash_bucket_strong

- Contents
- Aliases:

Converts each string in the input Tensor to its hash mod by a number of buckets.

Aliases:

- `tf.compat.v1.string_to_hash_bucket_strong`
- `tf.compat.v1.strings.to_hash_bucket_strong`
- `tf.compat.v2.strings.to_hash_bucket_strong`
- `tf.strings.to_hash_bucket_strong`

```
tf.strings.to_hash_bucket_strong(
    input,
    num_buckets,
    key,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

The hash function is deterministic on the content of the string within the process. The hash function is a keyed hash function, where attribute `key` defines the key of the hash function. `key` is an array of 2 elements.

A strong hash is important when inputs may be malicious, e.g. URLs with additional components. Adversaries could try to make their inputs hash to the same bucket for a denial-of-service attack or to skew the results. A strong hash can be used to make it difficult to find inputs with a skewed hash value distribution over buckets. This requires that the hash function is seeded by a high-entropy (random) "key" unknown to the adversary.

The additional robustness comes at a cost of roughly 4x higher compute time than `tf.string_to_hash_bucket_fast`.

Args:

- **`input`**: A `Tensor` of type `string`. The strings to assign a hash bucket.
- **`num_buckets`**: An `int` that is `>= 1`. The number of buckets.
- **`key`**: A list of `ints`. The key used to seed the hash function, passed as a list of two uint64 elements.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `int64`.

tf.strings.to_number

- Contents
- Aliases:
- Used in the guide:

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

Aliases:

- `tf.compat.v2.strings.to_number`
- `tf.strings.to_number`

```
tf.strings.to_number(
    input,
    out_type=tf.dtypes.float32,
    name=None
)
```

Defined in `python/ops/string_ops.py`.

Used in the guide:

- Using the SavedModel format

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

Args:

- **`input`**: A `Tensor` of type `string`.
- **`out_type`**: An optional `tf.DType` from: `tf.float32, tf.float64, tf.int32, tf.int64`. 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`.

tf.strings.unicode_decode

- Contents
- Aliases:
- Used in the tutorials:

Decodes each string in `input` into a sequence of Unicode code points.

Aliases:

- `tf.compat.v1.strings.unicode_decode`
- `tf.compat.v2.strings.unicode_decode`
- `tf.strings.unicode_decode`

```
tf.strings.unicode_decode(
    input,
    input_encoding,
    errors='replace',
    replacement_char=65533,
    replace_control_characters=False,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Used in the tutorials:

- Unicode strings

`result[i1...iN, j]` is the Unicode codepoint for the `j`th character in `input[i1...iN]`, when decoded using `input_encoding`.

Args:

- **`input`**: An `N` dimensional potentially ragged `string` tensor with shape `[D1...DN]`. `N` must be statically known.
- **`input_encoding`**: String name for the unicode encoding that should be used to decode each string.
- **`errors`**: Specifies the response when an input string can't be converted using the indicated encoding. One of:
- `'strict'`: Raise an exception for any illegal substrings.
- `'replace'`: Replace illegal substrings with `replacement_char`.
- `'ignore'`: Skip illegal substrings.
- **`replacement_char`**: The replacement codepoint to be used in place of invalid substrings in `input` when `errors='replace'`; and in place of C0 control characters in `input` when `replace_control_characters=True`.
- **`replace_control_characters`**: Whether to replace the C0 control characters `(U+0000 - U+001F)` with the `replacement_char`.
- **`name`**: A name for the operation (optional).

Returns:

A `N+1` dimensional `int32` tensor with shape `[D1...DN, (num_chars)]`. The returned tensor is a `tf.Tensor` if `input` is a scalar, or a `tf.RaggedTensor` otherwise.

Example:

```
>>> input = [s.encode('utf8') for s in (u'G\xf6\xf6dnight', u'\U0001f60a')]
>>> tf.strings.unicode_decode(input, 'UTF-8').tolist()
[[71, 246, 246, 100, 110, 105, 103, 104, 116], [128522]]
```

tf.strings.unicode_decode_with_offsets

- Contents
- Aliases:
- Used in the tutorials:

Decodes each string into a sequence of code points with start offsets.

Aliases:

- `tf.compat.v1.strings.unicode_decode_with_offsets`
- `tf.compat.v2.strings.unicode_decode_with_offsets`
- `tf.strings.unicode_decode_with_offsets`

```
tf.strings.unicode_decode_with_offsets(
    input,
    input_encoding,
    errors='replace',
    replacement_char=65533,
    replace_control_characters=False,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Used in the tutorials:

- Unicode strings

This op is similar to `tf.strings.decode(...)`, but it also returns the start offset for each character in its respective string. This information can be used to align the characters with the original byte sequence.

Returns a tuple `(codepoints, start_offsets)` where:

- `codepoints[i1...iN, j]` is the Unicode codepoint for the `j`th character in `input[i1...iN]`, when decoded using `input_encoding`.
- `start_offsets[i1...iN, j]` is the start byte offset for the `j`th character in `input[i1...iN]`, when decoded using `input_encoding`.

Args:

- **`input`**: An `N` dimensional potentially ragged `string` tensor with shape `[D1...DN]`. `N` must be statically known.
- **`input_encoding`**: String name for the unicode encoding that should be used to decode each string.
- **`errors`**: Specifies the response when an input string can't be converted using the indicated encoding. One of:
- `'strict'`: Raise an exception for any illegal substrings.
- `'replace'`: Replace illegal substrings with `replacement_char`.
- `'ignore'`: Skip illegal substrings.
- **`replacement_char`**: The replacement codepoint to be used in place of invalid substrings in `input` when `errors='replace'`; and in place of C0 control characters in `input` when `replace_control_characters=True`.
- **`replace_control_characters`**: Whether to replace the C0 control characters `(U+0000 - U+001F)` with the `replacement_char`.
- **`name`**: A name for the operation (optional).

Returns:

A tuple of `N+1` dimensional tensors `(codepoints, start_offsets)`.

- `codepoints` is an `int32` tensor with shape `[D1...DN, (num_chars)]`.
- `offsets` is an `int64` tensor with shape `[D1...DN, (num_chars)]`.

The returned tensors are `tf.Tensor`s if `input` is a scalar, or `tf.RaggedTensor`s otherwise.

Example:

```
>>> input = [s.encode('utf8') for s in (u'G\xf6\xf6dnight', u'\U0001f60a')]
>>> result = tf.strings.unicode_decode_with_offsets(input, 'UTF-8')
>>> result[0].tolist()  # codepoints
[[71, 246, 246, 100, 110, 105, 103, 104, 116], [128522]]
>>> result[1].tolist()  # offsets
[[0, 1, 3, 5, 6, 7, 8, 9, 10], [0]]
```

tf.strings.unicode_encode

- Contents
- Aliases:
- Used in the tutorials:

Encodes each sequence of Unicode code points in `input` into a string.

Aliases:

- `tf.compat.v1.strings.unicode_encode`
- `tf.compat.v2.strings.unicode_encode`
- `tf.strings.unicode_encode`

```
tf.strings.unicode_encode(
    input,
    output_encoding,
    errors='replace',
    replacement_char=65533,
    name=None
```

```
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Used in the tutorials:

- Unicode strings

`result[i1...iN]` is the string formed by concatenating the Unicode codepoints `input[1...iN, :]`, encoded using `output_encoding`.

Args:

- **`input`**: An `N+1` dimensional potentially ragged integer tensor with shape `[D1...DN, num_chars]`.
- **`output_encoding`**: Unicode encoding that should be used to encode each codepoint sequence. Can be `"UTF-8"`, `"UTF-16-BE"`, or `"UTF-32-BE"`.
- **`errors`**: Specifies the response when an invalid codepoint is encountered (optional). One of: * `'replace'`: Replace invalid codepoint with the `replacement_char`. (default) * `'ignore'`: Skip invalid codepoints. * `'strict'`: Raise an exception for any invalid codepoint.
- **`replacement_char`**: The replacement character codepoint to be used in place of any invalid input when `errors='replace'`. Any valid unicode codepoint may be used. The default value is the default unicode replacement character which is 0xFFFD (U+65533).
- **`name`**: A name for the operation (optional).

Returns:

A `N` dimensional `string` tensor with shape `[D1...DN]`.

Example:

```
>>> input = [[71, 246, 246, 100, 110, 105, 103, 104, 116], [128522]]
>>> unicode_encode(input, 'UTF-8')
['G\xc3\xb6\xc3\xb6dnight', '\xf0\x9f\x98\x8a']
```

tf.strings.unicode_script

- Contents
- Aliases:
- Used in the tutorials:

Determine the script codes of a given tensor of Unicode integer code points.

Aliases:

- `tf.compat.v1.strings.unicode_script`
- `tf.compat.v2.strings.unicode_script`
- `tf.strings.unicode_script`

```
tf.strings.unicode_script(
    input,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Used in the tutorials:

- Unicode strings

This operation converts Unicode code points to script codes corresponding to each code point. Script codes correspond to International Components for Unicode (ICU) UScriptCode values. See http://icu-project.org/apiref/icu4c/uscript_8h.html. Returns -1 (USCRIPT_INVALID_CODE) for invalid codepoints. Output shape will match input shape.

Args:

- **`input`**: A `Tensor` of type `int32`. A Tensor of int32 Unicode code points.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `int32`.

tf.strings.unicode_split

- Contents
- Aliases:
- Used in the tutorials:

Splits each string in `input` into a sequence of Unicode code points.

Aliases:

- `tf.compat.v1.strings.unicode_split`
- `tf.compat.v2.strings.unicode_split`
- `tf.strings.unicode_split`

```
tf.strings.unicode_split(
    input,
    input_encoding,
    errors='replace',
    replacement_char=65533,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

Used in the tutorials:

- Unicode strings

`result[i1...iN, j]` is the substring of `input[i1...iN]` that encodes its `j`th character, when decoded using `input_encoding`.

Args:

- **`input`**: An `N` dimensional potentially ragged `string` tensor with shape `[D1...DN]`. `N` must be statically known.
- **`input_encoding`**: String name for the unicode encoding that should be used to decode each string.
- **`errors`**: Specifies the response when an input string can't be converted using the indicated encoding. One of:
- `'strict'`: Raise an exception for any illegal substrings.
- `'replace'`: Replace illegal substrings with `replacement_char`.
- `'ignore'`: Skip illegal substrings.
- **`replacement_char`**: The replacement codepoint to be used in place of invalid substrings in `input` when `errors='replace'`.
- **`name`**: A name for the operation (optional).

Returns:

A `N+1` dimensional `int32` tensor with shape `[D1...DN, (num_chars)]`. The returned tensor is a `tf.Tensor` if `input` is a scalar, or a `tf.RaggedTensor` otherwise.

Example:

```
>>> input = [s.encode('utf8') for s in (u'G\xf6\xf6dnight', u'\U0001f60a')]
>>> tf.strings.unicode_split(input, 'UTF-8').tolist()
```

```
[['G', '\xc3\xb6', '\xc3\xb6', 'd', 'n', 'i', 'g', 'h', 't'],
 ['\xf0\x9f\x98\x8a']]
```

tf.strings.unicode_split_with_offsets

- Contents
- Aliases:

Splits each string into a sequence of code points with start offsets.

Aliases:

- `tf.compat.v1.strings.unicode_split_with_offsets`
- `tf.compat.v2.strings.unicode_split_with_offsets`
- `tf.strings.unicode_split_with_offsets`

```
tf.strings.unicode_split_with_offsets(
    input,
    input_encoding,
    errors='replace',
    replacement_char=65533,
    name=None
)
```

Defined in `python/ops/ragged/ragged_string_ops.py`.

This op is similar to `tf.strings.decode(...)`, but it also returns the start offset for each character in its respective string. This information can be used to align the characters with the original byte sequence.

Returns a tuple `(chars, start_offsets)` where:

- `chars[i1...iN, j]` is the substring of `input[i1...iN]` that encodes its `j`th character, when decoded using `input_encoding`.
- `start_offsets[i1...iN, j]` is the start byte offset for the `j`th character in `input[i1...iN]`, when decoded using `input_encoding`.

Args:

- **`input`**: An `N` dimensional potentially ragged `string` tensor with shape `[D1...DN]`. `N` must be statically known.
- **`input_encoding`**: String name for the unicode encoding that should be used to decode each string.
- **`errors`**: Specifies the response when an input string can't be converted using the indicated encoding. One of:
- `'strict'`: Raise an exception for any illegal substrings.
- `'replace'`: Replace illegal substrings with `replacement_char`.
- `'ignore'`: Skip illegal substrings.
- **`replacement_char`**: The replacement codepoint to be used in place of invalid substrings in `input` when `errors='replace'`.
- **`name`**: A name for the operation (optional).

Returns:

A tuple of `N+1` dimensional tensors `(codepoints, start_offsets)`.

- `codepoints` is an `int32` tensor with shape `[D1...DN, (num_chars)]`.
- `offsets` is an `int64` tensor with shape `[D1...DN, (num_chars)]`.

The returned tensors are `tf.Tensor`s if `input` is a scalar, or `tf.RaggedTensor`s otherwise.

Example:

```
>>> input = [s.encode('utf8') for s in (u'G\xf6\xf6dnight', u'\U0001f60a')]
>>> result = tf.strings.unicode_split_with_offsets(input, 'UTF-8')
```

```
>>> result[0].tolist()  # character substrings
[['G', '\xc3\xb6', '\xc3\xb6', 'd', 'n', 'i', 'g', 'h', 't'],
 ['\xf0\x9f\x98\x8a']]
>>> result[1].tolist()  # offsets
[0, 1, 3, 5, 6, 7, 8, 9, 10], [0]]
```

tf.strings.unicode_transcode

- Contents
- Aliases:
- Used in the tutorials:

Transcode the input text from a source encoding to a destination encoding.

Aliases:

- `tf.compat.v1.strings.unicode_transcode`
- `tf.compat.v2.strings.unicode_transcode`
- `tf.strings.unicode_transcode`

```
tf.strings.unicode_transcode(
    input,
    input_encoding,
    output_encoding,
    errors='replace',
    replacement_char=65533,
    replace_control_characters=False,
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Used in the tutorials:

- Unicode strings

The input is a string tensor of any shape. The output is a string tensor of the same shape containing the transcoded strings. Output strings are always valid unicode. If the input contains invalid encoding positions, the `errors` attribute sets the policy for how to deal with them. If the default error-handling policy is used, invalid formatting will be substituted in the output by the `replacement_char`. If the errors policy is to `ignore`, any invalid encoding positions in the input are skipped and not included in the output. If it set to `strict` then any invalid formatting will result in an InvalidArgument error.

This operation can be used with `output_encoding = input_encoding` to enforce correct formatting for inputs even if they are already in the desired encoding.

If the input is prefixed by a Byte Order Mark needed to determine encoding (e.g. if the encoding is UTF-16 and the BOM indicates big-endian), then that BOM will be consumed and not emitted into the output. If the input encoding is marked with an explicit endianness (e.g. UTF-16-BE), then the BOM is interpreted as a non-breaking-space and is preserved in the output (including always for UTF-8).

The end result is that if the input is marked as an explicit endianness the transcoding is faithful to all codepoints in the source. If it is not marked with an explicit endianness, the BOM is not considered part of the string itself but as metadata, and so is not preserved in the output.

Args:

- **`input`**: A `Tensor` of type `string`. The text to be processed. Can have any shape.
- **`input_encoding`**: A `string`. Text encoding of the input strings. This is any of the encodings supported by ICU ucnv algorithmic converters. Examples: `"UTF-16", "US ASCII", "UTF-8"`.

- **`output_encoding`**: A `string` from: `"UTF-8", "UTF-16-BE", "UTF-32-BE"`. The unicode encoding to use in the output. Must be one of `"UTF-8", "UTF-16-BE", "UTF-32-BE"`. Multi-byte encodings will be big-endian.
- **`errors`**: An optional `string` from: `"strict", "replace", "ignore"`. Defaults to `"replace"`. Error handling policy when there is invalid formatting found in the input. The value of 'strict' will cause the operation to produce a InvalidArgument error on any invalid input formatting. A value of 'replace' (the default) will cause the operation to replace any invalid formatting in the input with the `replacement_char` codepoint. A value of 'ignore' will cause the operation to skip any invalid formatting in the input and produce no corresponding output character.
- **`replacement_char`**: An optional `int`. Defaults to `65533`. The replacement character codepoint to be used in place of any invalid formatting in the input when `errors='replace'`. Any valid unicode codepoint may be used. The default value is the default unicode replacement character is 0xFFFD or U+65533.)
  Note that for UTF-8, passing a replacement character expressible in 1 byte, such as ' ', will preserve string alignment to the source since invalid bytes will be replaced with a 1-byte replacement. For UTF-16-BE and UTF-16-LE, any 1 or 2 byte replacement character will preserve byte alignment to the source.
- **`replace_control_characters`**: An optional `bool`. Defaults to `False`. Whether to replace the C0 control characters (00-1F) with the `replacement_char`. Default is false.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `string`.

tf.strings.upper

- Contents
- Aliases:

TODO: add doc.

Aliases:

- `tf.compat.v1.strings.upper`
- `tf.compat.v2.strings.upper`
- `tf.strings.upper`

```
tf.strings.upper(
    input,
    encoding='',
    name=None
)
```

Defined in generated file: `python/ops/gen_string_ops.py`.

Args:

- **`input`**: A `Tensor` of type `string`.
- **`encoding`**: An optional `string`. Defaults to `""`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `string`.

Module: tf.summary

- **Contents**
- Aliases:
- Modules

- Classes
- Functions

Operations for writing summary data, for use in analysis and visualization.

Aliases:

- Module `tf.compat.v2.summary`
- Module `tf.summary`

Defined in `summary/_tf/summary/__init__.py`.

The `tf.summary` module provides APIs for writing summary data. This data can be visualized in TensorBoard, the visualization toolkit that comes with TensorFlow. See the TensorBoard website for more detailed tutorials about how to use these APIs, or some quick examples below.

Example usage with eager execution, the default in TF 2.0:

```
writer = tf.summary.create_file_writer("/tmp/mylogs")
with writer.as_default():
  for step in range(100):
    # other model code would go here
    tf.summary.scalar("my_metric", 0.5, step=step)
    writer.flush()
```

Example usage with `tf.function` graph execution:

```
writer = tf.summary.create_file_writer("/tmp/mylogs")

@tf.function
def my_func(step):
  # other model code would go here
  with writer.as_default():
    tf.summary.scalar("my_metric", 0.5, step=step)

for step in range(100):
  my_func(step)
  writer.flush()
```

Example usage with legacy TF 1.x graph execution:

```
with tf.compat.v1.Graph().as_default():
  step = tf.Variable(0, dtype=tf.int64)
  step_update = step.assign_add(1)
  writer = tf.summary.create_file_writer("/tmp/mylogs")
  with writer.as_default():
    tf.summary.scalar("my_metric", 0.5, step=step)
  all_summary_ops = tf.compat.v1.summary.all_v2_summary_ops()
  writer_flush = writer.flush()

  sess = tf.compat.v1.Session()
  sess.run([writer.init(), step.initializer])
  for i in range(100):
    sess.run(all_summary_ops)
    sess.run(step_update)
    sess.run(writer_flush)
```

Modules

`experimental` module: Public API for tf.summary.experimental namespace.

Classes

`class SummaryWriter`: Interface representing a stateful summary writer object.

Functions

`audio(...)`: Write an audio summary.
`create_file_writer(...)`: Creates a summary file writer for the given log directory.
`create_noop_writer(...)`: Returns a summary writer that does nothing.
`flush(...)`: Forces summary writer to send any buffered data to storage.
`histogram(...)`: Write a histogram summary.
`image(...)`: Write an image summary.
`record_if(...)`: Sets summary recording on or off per the provided boolean value.
`scalar(...)`: Write a scalar summary.
`text(...)`: Write a text summary.
`trace_export(...)`: Stops and exports the active trace as a Summary and/or profile file.
`trace_off(...)`: Stops the current trace and discards any collected information.
`trace_on(...)`: Starts a trace to record computation graphs and profiling information.
`write(...)`: Writes a generic summary to the default SummaryWriter if one exists.

tf.summary.audio

- **Contents**
- Aliases:

Write an audio summary.

Aliases:

- `tf.compat.v2.summary.audio`
- `tf.summary.audio`

```
tf.summary.audio(
    name,
    data,
    sample_rate,
    step=None,
    max_outputs=3,
    encoding=None,
    description=None
)
```

Defined in `plugins/audio/summary_v2.py`.

Arguments:

- **`name`**: A name for this summary. The summary tag used for TensorBoard will be this name prefixed by any active name scopes.
- **`data`**: A `Tensor` representing audio data with shape `[k, t, c]`, where `k` is the number of audio clips, `t` is the number of frames, and `c` is the number of channels. Elements should be floating-point values in `[-1.0, 1.0]`. Any of the dimensions may be statically unknown (i.e., `None`).
- **`sample_rate`**: An `int` or rank-0 `int32` `Tensor` that represents the sample rate, in Hz. Must be positive.
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.

- `max_outputs`: Optional `int` or rank-0 integer `Tensor`. At most this many audio clips will be emitted at each step. When more than `max_outputs` many clips are provided, the first `max_outputs` many clips will be used and the rest silently discarded.
- `encoding`: Optional constant `str` for the desired encoding. Only "wav" is currently supported, but this is not guaranteed to remain the default, so if you want "wav" in particular, set this explicitly.
- `description`: Optional long-form description for this summary, as a constant `str`. Markdown is supported. Defaults to empty.

Returns:
True on success, or false if no summary was emitted because no default summary writer was available.

Raises:
- `ValueError`: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.create_file_writer

- **Contents**
- Aliases:

Creates a summary file writer for the given log directory.

Aliases:
- `tf.compat.v2.summary.create_file_writer`
- `tf.summary.create_file_writer`

```
tf.summary.create_file_writer(
    logdir,
    max_queue=None,
    flush_millis=None,
    filename_suffix=None,
    name=None
)
```

Defined in `python/ops/summary_ops_v2.py`.

Args:
- `logdir`: a string specifying the directory in which to write an event file.
- `max_queue`: the largest number of summaries to keep in a queue; will flush once the queue gets bigger than this. Defaults to 10.
- `flush_millis`: the largest interval between flushes. Defaults to 120,000.
- `filename_suffix`: optional suffix for the event file name. Defaults to `.v2`.
- `name`: a name for the op that creates the writer.

Returns:
A SummaryWriter object.

tf.summary.create_noop_writer

- **Contents**
- Aliases:

Returns a summary writer that does nothing.

Aliases:
- `tf.compat.v2.summary.create_noop_writer`
- `tf.summary.create_noop_writer`

```
tf.summary.create_noop_writer()
```

Defined in `python/ops/summary_ops_v2.py`.
This is useful as a placeholder in code that expects a context manager.

tf.summary.flush

- **Contents**
- Aliases:

Forces summary writer to send any buffered data to storage.

Aliases:

- `tf.compat.v2.summary.flush`
- `tf.summary.flush`

```
tf.summary.flush(
    writer=None,
    name=None
)
```

Defined in `python/ops/summary_ops_v2.py`.
This operation blocks until that finishes.

Args:

- **`writer`**: The `tf.summary.SummaryWriter` resource to flush. The thread default will be used if this parameter is None. Otherwise a `tf.no_op` is returned.
- **`name`**: A name for the operation (optional).

Returns:
The created `tf.Operation`.

tf.summary.histogram

- **Contents**
- Aliases:

Write a histogram summary.

Aliases:

- `tf.compat.v2.summary.histogram`
- `tf.summary.histogram`

```
tf.summary.histogram(
    name,
    data,
    step=None,
    buckets=None,
    description=None
)
```

Defined in `plugins/histogram/summary_v2.py`.

Arguments:

- **`name`**: A name for this summary. The summary tag used for TensorBoard will be this name prefixed by any active name scopes.
- **`data`**: A `Tensor` of any shape. Must be castable to `float64`.

- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- **`buckets`**: Optional positive `int`. The output will have this many buckets, except in two edge cases. If there is no data, then there are no buckets. If there is data but all points have the same value, then there is one bucket whose left and right endpoints are the same.
- **`description`**: Optional long-form description for this summary, as a constant `str`. Markdown is supported. Defaults to empty.

Returns:
True on success, or false if no summary was emitted because no default summary writer was available.

Raises:
- **`ValueError`**: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.image

- **Contents**
- Aliases:

Write an image summary.

Aliases:
- `tf.compat.v2.summary.image`
- `tf.summary.image`

```
tf.summary.image(
    name,
    data,
    step=None,
    max_outputs=3,
    description=None
)
```

Defined in `plugins/image/summary_v2.py`.

Arguments:
- **`name`**: A name for this summary. The summary tag used for TensorBoard will be this name prefixed by any active name scopes.
- **`data`**: A `Tensor` representing pixel data with shape `[k, h, w, c]`, where `k` is the number of images, `h` and `w` are the height and width of the images, and `c` is the number of channels, which should be 1, 2, 3, or 4 (grayscale, grayscale with alpha, RGB, RGBA). Any of the dimensions may be statically unknown (i.e., `None`). Floating point data will be clipped to the range [0,1).
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- **`max_outputs`**: Optional `int` or rank-0 integer `Tensor`. At most this many images will be emitted at each step. When more than `max_outputs` many images are provided, the first `max_outputs` many images will be used and the rest silently discarded.
- **`description`**: Optional long-form description for this summary, as a constant `str`. Markdown is supported. Defaults to empty.

Returns:
True on success, or false if no summary was emitted because no default summary writer was available.

Raises:

- `ValueError`: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.record_if

- **Contents**
- Aliases:

Sets summary recording on or off per the provided boolean value.

Aliases:

- `tf.compat.v2.summary.record_if`
- `tf.summary.record_if`

```
tf.summary.record_if(condition)
```

Defined in `python/ops/summary_ops_v2.py`.

The provided value can be a python boolean, a scalar boolean Tensor, or or a callable providing such a value; if a callable is passed it will be invoked on-demand to determine whether summary writing will occur.

Args:

- `condition`: can be True, False, a bool Tensor, or a callable providing such.

Yields:

Returns a context manager that sets this value on enter and restores the previous value on exit.

tf.summary.scalar

- **Contents**
- Aliases:

Write a scalar summary.

Aliases:

- `tf.compat.v2.summary.scalar`
- `tf.summary.scalar`

```
tf.summary.scalar(
    name,
    data,
    step=None,
    description=None
)
```

Defined in `plugins/scalar/summary_v2.py`.

Arguments:

- `name`: A name for this summary. The summary tag used for TensorBoard will be this name prefixed by any active name scopes.
- `data`: A real numeric scalar value, convertible to a `float32` Tensor.
- `step`: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- `description`: Optional long-form description for this summary, as a constant `str`. Markdown is supported. Defaults to empty.

Returns:
True on success, or false if no summary was written because no default summary writer was available.

Raises:

- `ValueError`: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.SummaryWriter

Contents

- Class SummaryWriter
  - Aliases:
- Methods
  - as_default

Class `SummaryWriter`

Interface representing a stateful summary writer object.

Aliases:

- Class `tf.compat.v2.summary.SummaryWriter`
- Class `tf.summary.SummaryWriter`

Defined in `python/ops/summary_ops_v2.py`.

Methods

`as_default`

```
as_default()
```

Returns a context manager that enables summary writing.

`close`

```
close()
```

Flushes and closes the summary writer.

`flush`

```
flush()
```

Flushes any buffered data.

`init`

```
init()
```

Initializes the summary writer.

`set_as_default`

```
set_as_default()
```

Enables this summary writer for the current thread.

tf.summary.text

Contents

- Aliases:

Write a text summary.

Aliases:

- `tf.compat.v2.summary.text`
- `tf.summary.text`

```
tf.summary.text(
    name,
    data,
    step=None,
    description=None
)
```

Defined in `plugins/text/summary_v2.py`.

Arguments:

- **`name`**: A name for this summary. The summary tag used for TensorBoard will be this name prefixed by any active name scopes.
- **`data`**: A UTF-8 string tensor value.
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- **`description`**: Optional long-form description for this summary, as a constant `str`. Markdown is supported. Defaults to empty.

Returns:

True on success, or false if no summary was emitted because no default summary writer was available.

Raises:

- **`ValueError`**: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.trace_export

- **Contents**
- Aliases:

Stops and exports the active trace as a Summary and/or profile file.

Aliases:

- `tf.compat.v2.summary.trace_export`
- `tf.summary.trace_export`

```
tf.summary.trace_export(
    name,
    step=None,
    profiler_outdir=None
)
```

Defined in `python/ops/summary_ops_v2.py`.

Stops the trace and exports all metadata collected during the trace to the default SummaryWriter, if one has been set.

Args:

- **`name`**: A name for the summary to be written.
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.

- `profiler_outdir`: Output directory for profiler. It is required when profiler is enabled when trace was started. Otherwise, it is ignored.

Raises:

- `ValueError`: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

tf.summary.trace_off

- **Contents**
- Aliases:

Stops the current trace and discards any collected information.

Aliases:

- `tf.compat.v2.summary.trace_off`
- `tf.summary.trace_off`

```
tf.summary.trace_off()
```

Defined in `python/ops/summary_ops_v2.py`.

tf.summary.trace_on

- **Contents**
- Aliases:

Starts a trace to record computation graphs and profiling information.

Aliases:

- `tf.compat.v2.summary.trace_on`
- `tf.summary.trace_on`

```
tf.summary.trace_on(
    graph=True,
    profiler=False
)
```

Defined in `python/ops/summary_ops_v2.py`.

Must be invoked in eager mode.

When enabled, TensorFlow runtime will collection information that can later be exported and consumed by TensorBoard. The trace is activated across the entire TensorFlow runtime and affects all threads of execution.

To stop the trace and export the collected information, use `tf.summary.trace_export`. To stop the trace without exporting, use `tf.summary.trace_off`.

Args:

- `graph`: If True, enables collection of executed graphs. It includes ones from tf.function invocation and ones from the legacy graph mode. The default is True.
- `profiler`: If True, enables the advanced profiler. Enabling profiler implicitly enables the graph collection. The profiler may incur a high memory overhead. The default is False.

tf.summary.write

- **Contents**
- Aliases:

Writes a generic summary to the default SummaryWriter if one exists.

Aliases:

- `tf.compat.v2.summary.write`
- `tf.summary.write`

```
tf.summary.write(
    tag,
    tensor,
    step=None,
    metadata=None,
    name=None
)
```

Defined in `python/ops/summary_ops_v2.py`.

This exists primarily to support the definition of type-specific summary ops like scalar() and image(), and is not intended for direct use unless defining a new type-specific summary op.

Args:

- **`tag`**: string tag used to identify the summary (e.g. in TensorBoard), usually generated with `tf.summary.summary_scope`
- **`tensor`**: the Tensor holding the summary data to write
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- **`metadata`**: Optional SummaryMetadata, as a proto or serialized bytes
- **`name`**: Optional string name for this op.

Returns:

True on success, or false if no summary was written because no default summary writer was available.

Raises:

- **`ValueError`**: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

Module: tf.summary.experimental

- **Contents**
- Aliases:
- Functions

Public API for tf.summary.experimental namespace.

Aliases:

- Module `tf.compat.v2.summary.experimental`
- Module `tf.summary.experimental`

Functions

`get_step(...)`: Returns the default summary step for the current thread.

`set_step(...)`: Sets the default summary step for the current thread.

`summary_scope(...)`: Experimental context manager for use when defining a custom summary op.

`write_raw_pb(...)`: Writes a summary using raw `tf.compat.v1.Summary` protocol buffers.

tf.summary.experimental.get_step

- **Contents**
- Aliases:

Returns the default summary step for the current thread.

Aliases:

- `tf.compat.v2.summary.experimental.get_step`
- `tf.summary.experimental.get_step`

```
tf.summary.experimental.get_step()

```

Defined in `python/ops/summary_ops_v2.py`.

Returns:

The step set by `tf.summary.experimental.set_step()` if one has been set, otherwise None.

tf.summary.experimental.set_step

- **Contents**
- Aliases:

Sets the default summary step for the current thread.

Aliases:

- `tf.compat.v2.summary.experimental.set_step`
- `tf.summary.experimental.set_step`

```
tf.summary.experimental.set_step(step)

```

Defined in `python/ops/summary_ops_v2.py`.

For convenience, this function sets a default value for the `step` parameter used in summary-writing functions elsewhere in the API so that it need not be explicitly passed in every such invocation. The value can be a constant or a variable, and can be retrieved via `tf.summary.experimental.get_step()`.

Note: when using this with @tf.functions, the step value will be captured at the time the function is traced, so changes to the step outside the function will not be reflected inside the function unless using a `tf.Variable`step.

Args:

- **`step`**: An `int64`-castable default step value, or None to unset.

tf.summary.experimental.summary_scope

- **Contents**
- Aliases:

Experimental context manager for use when defining a custom summary op.

Aliases:

- `tf.compat.v2.summary.experimental.summary_scope`
- `tf.summary.experimental.summary_scope`

```
tf.summary.experimental.summary_scope(
    name,
    default_name='summary',
    values=None
)

```

Defined in `python/ops/summary_ops_v2.py`.

This behaves similarly to `tf.name_scope`, except that it returns a generated summary tag in addition to the scope name. The tag is structurally similar to the scope name - derived from the user-provided name, prefixed with enclosing name scopes if any - but we relax the constraint that it be uniquified,

as well as the character set limitation (so the user-provided name can contain characters not legal for scope names; in the scope name these are removed).
This makes the summary tag more predictable and consistent for the user.
For example, to define a new summary op called `my_op`:

```
def my_op(name, my_value, step):
  with tf.summary.summary_scope(name, "MyOp", [my_value]) as (tag, scope):
    my_value = tf.convert_to_tensor(my_value)
    return tf.summary.write(tag, my_value, step=step)
```

Args:

- **`name`**: string name for the summary.
- **`default_name`**: Optional; if provided, used as default name of the summary.
- **`values`**: Optional; passed as `values` parameter to name_scope.

Yields:
A tuple `(tag, scope)` as described above.

tf.summary.experimental.write_raw_pb

- **Contents**
- Aliases:

Writes a summary using raw `tf.compat.v1.Summary` protocol buffers.

Aliases:

- `tf.compat.v2.summary.experimental.write_raw_pb`
- `tf.summary.experimental.write_raw_pb`

```
tf.summary.experimental.write_raw_pb(
    tensor,
    step=None,
    name=None
)
```

Defined in `python/ops/summary_ops_v2.py`.
Experimental: this exists to support the usage of V1-style manual summary writing (via the construction of a `tf.compat.v1.Summary` protocol buffer) with the V2 summary writing API.

Args:

- **`tensor`**: the string Tensor holding one or more serialized `Summary` protobufs
- **`step`**: Explicit `int64`-castable monotonic step value for this summary. If omitted, this defaults to `tf.summary.experimental.get_step()`, which must not be None.
- **`name`**: Optional string name for this op.

Returns:
True on success, or false if no summary was written because no default summary writer was available.

Raises:

- **`ValueError`**: if a default writer exists, but no step was provided and`tf.summary.experimental.get_step()` is None.

Module: tf.compat.v1.summary

- **Contents**
- Classes
- Functions

Operations for writing summary data, for use in analysis and visualization.
See the Summaries and TensorBoard guide.

Classes

`class Event`
`class FileWriter`: Writes `Summary` protocol buffers to event files.
`class FileWriterCache`: Cache for file writers.
`class SessionLog`
`class Summary`
`class SummaryDescription`
`class TaggedRunMetadata`

Functions

`all_v2_summary_ops(...)`: Returns all V2-style summary ops defined in the current default graph.
`audio(...)`: Outputs a `Summary` protocol buffer with audio.
`get_summary_description(...)`: Given a TensorSummary node_def, retrieve its SummaryDescription.
`histogram(...)`: Outputs a `Summary` protocol buffer with a histogram.
`image(...)`: Outputs a `Summary` protocol buffer with images.
`initialize(...)`: Initializes summary writing for graph execution mode.
`merge(...)`: Merges summaries.
`merge_all(...)`: Merges all summaries collected in the default graph.
`scalar(...)`: Outputs a `Summary` protocol buffer containing a single scalar value.
`tensor_summary(...)`: Outputs a `Summary` protocol buffer with a serialized tensor.proto.
`text(...)`: Summarizes textual data.

tf.compat.v1.summary.all_v2_summary_ops

Returns all V2-style summary ops defined in the current default graph.

```
tf.compat.v1.summary.all_v2_summary_ops()
```

Defined in `python/ops/summary_ops_v2.py`.
This includes ops from TF 2.0 tf.summary and TF 1.x tf.contrib.summary (except for `tf.contrib.summary.graph` and `tf.contrib.summary.import_event`), but does *not* include TF 1.x tf.summary ops.

Returns:
List of summary ops, or None if called under eager execution.

tf.compat.v1.summary.audio

Outputs a `Summary` protocol buffer with audio.

```
tf.compat.v1.summary.audio(
    name,
    tensor,
    sample_rate,
    max_outputs=3,
    collections=None,
    family=None
)
```

Defined in `python/summary/summary.py`.
The summary has up to `max_outputs` summary values containing audio. The audio is built from `tensor` which must be 3-D with shape `[batch_size, frames, channels]` or 2-D with

shape `[batch_size, frames]`. The values are assumed to be in the range of `[-1.0, 1.0]` with a sample rate of `sample_rate`.
The `tag` in the outputted Summary.Value protobufs is generated based on the name, with a suffix depending on the max_outputs setting:

- If `max_outputs` is 1, the summary value tag is '*name*/audio'.
- If `max_outputs` is greater than 1, the summary value tags are generated sequentially as '*name*/audio/0', '*name*/audio/1', etc

Args:

- **`name`**: A name for the generated node. Will also serve as a series name in TensorBoard.
- **`tensor`**: A 3-D `float32` `Tensor` of shape `[batch_size, frames, channels]` or a 2-D `float32` `Tensor` of shape `[batch_size, frames]`.
- **`sample_rate`**: A Scalar `float32` `Tensor` indicating the sample rate of the signal in hertz.
- **`max_outputs`**: Max number of batch elements to generate audio for.
- **`collections`**: Optional list of ops.GraphKeys. The collections to add the summary to. Defaults to [_ops.GraphKeys.SUMMARIES]
- **`family`**: Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns:
A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

tf.compat.v1.summary.FileWriter

- **Contents**
- Class FileWriter
- __init__
- Methods
  - __enter__

Class `FileWriter`

Writes `Summary` protocol buffers to event files.
Defined in `python/summary/writer/writer.py`.
The `FileWriter` class provides a mechanism to create an event file in a given directory and add summaries and events to it. The class updates the file contents asynchronously. This allows a training program to call methods to add data to the file directly from the training loop, without slowing down training.
When constructed with a `tf.compat.v1.Session` parameter, a `FileWriter` instead forms a compatibility layer over new graph-based summaries (`tf.contrib.summary`) to facilitate the use of new summary writing with pre-existing code that expects a `FileWriter` instance.

`__init__`

```
__init__(
    logdir,
    graph=None,
    max_queue=10,
    flush_secs=120,
    graph_def=None,
    filename_suffix=None,
    session=None
)
```

Creates a `FileWriter`, optionally shared within the given session.

Typically, constructing a file writer creates a new event file in `logdir`. This event file will contain `Event` protocol buffers constructed when you call one of the following functions: `add_summary()`, `add_session_log()`, `add_event()`, or `add_graph()`.
If you pass a `Graph` to the constructor it is added to the event file. (This is equivalent to calling `add_graph()` later).
TensorBoard will pick the graph from the file and display it graphically so you can interactively explore the graph you built. You will usually pass the graph from the session in which you launched it:

```
...create a graph...
# Launch the graph in a session.
sess = tf.compat.v1.Session()
# Create a summary writer, add the 'graph' to the event file.
writer = tf.compat.v1.summary.FileWriter(<some-directory>, sess.graph)
```

The `session` argument to the constructor makes the returned `FileWriter` a compatibility layer over new graph-based summaries (`tf.contrib.summary`). Crucially, this means the underlying writer resource and events file will be shared with any other `FileWriter` using the same `session` and `logdir`, and with any `tf.contrib.summary.SummaryWriter` in this session using the the same shared resource name (which by default scoped to the logdir). If no such resource exists, one will be created using the remaining arguments to this constructor, but if one already exists those arguments are ignored. In either case, ops will be added to `session.graph` to control the underlying file writer resource. See `tf.contrib.summary` for more details.

Args:

- **`logdir`**: A string. Directory where event file will be written.
- **`graph`**: A `Graph` object, such as `sess.graph`.
- **`max_queue`**: Integer. Size of the queue for pending events and summaries.
- **`flush_secs`**: Number. How often, in seconds, to flush the pending events and summaries to disk.
- **`graph_def`**: DEPRECATED: Use the `graph` argument instead.
- **`filename_suffix`**: A string. Every event file's name is suffixed with `suffix`.
- **`session`**: A `tf.compat.v1.Session` object. See details above.

Raises:

- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility

`FileWriter` is not compatible with eager execution. To write TensorBoard summaries under eager execution, use `tf.contrib.summary` instead.

Methods

`__enter__`

```
__enter__()
```

Make usable with "with" statement.

`__exit__`

```
__exit__(
    unused_type,
    unused_value,
    unused_traceback
)
```

Make usable with "with" statement.

add_event

```
add_event(event)
```

Adds an event to the event file.

Args:

- **`event`**: An `Event` protocol buffer.

add_graph

```
add_graph(
    graph,
    global_step=None,
    graph_def=None
)
```

Adds a `Graph` to the event file.
The graph described by the protocol buffer will be displayed by TensorBoard. Most users pass a graph in the constructor instead.

Args:

- **`graph`**: A `Graph` object, such as `sess.graph`.
- **`global_step`**: Number. Optional global step counter to record with the graph.
- **`graph_def`**: DEPRECATED. Use the `graph` parameter instead.

Raises:

- **`ValueError`**: If both graph and graph_def are passed to the method.

add_meta_graph

```
add_meta_graph(
    meta_graph_def,
    global_step=None
)
```

Adds a `MetaGraphDef` to the event file.
The `MetaGraphDef` allows running the given graph via `saver.import_meta_graph()`.

Args:

- **`meta_graph_def`**: A `MetaGraphDef` object, often as returned by`saver.export_meta_graph()`.
- **`global_step`**: Number. Optional global step counter to record with the graph.

Raises:

- **`TypeError`**: If both `meta_graph_def` is not an instance of `MetaGraphDef`.

add_run_metadata

```
add_run_metadata(
    run_metadata,
    tag,
    global_step=None
)
```

Adds a metadata information for a single session.run() call.

Args:

- **`run_metadata`**: A `RunMetadata` protobuf object.
- **`tag`**: The tag name for this metadata.
- **`global_step`**: Number. Optional global step counter to record with the StepStats.

Raises:

- **`ValueError`**: If the provided tag was already used for this type of event.

`add_session_log`

```
add_session_log(
    session_log,
    global_step=None
)
```

Adds a `SessionLog` protocol buffer to the event file.
This method wraps the provided session in an `Event` protocol buffer and adds it to the event file.

Args:

- **`session_log`**: A `SessionLog` protocol buffer.
- **`global_step`**: Number. Optional global step value to record with the summary.

`add_summary`

```
add_summary(
    summary,
    global_step=None
)
```

Adds a `Summary` protocol buffer to the event file.
This method wraps the provided summary in an `Event` protocol buffer and adds it to the event file.
You can pass the result of evaluating any summary op, using `tf.Session.run` or `tf.Tensor.eval`, to this function. Alternatively, you can pass a `tf.compat.v1.Summary` protocol buffer that you populate with your own data. The latter is commonly done to report evaluation results in event files.

Args:

- **`summary`**: A `Summary` protocol buffer, optionally serialized as a string.
- **`global_step`**: Number. Optional global step value to record with the summary.

`close`

```
close()
```

Flushes the event file to disk and close the file.
Call this method when you do not need the summary writer anymore.

`flush`

```
flush()
```

Flushes the event file to disk.
Call this method to make sure that all pending events have been written to disk.

`get_logdir`

```
get_logdir()
```

Returns the directory where event file will be written.

reopen

```
reopen()
```

Reopens the EventFileWriter.
Can be called after `close()` to add more events in the same directory. The events will go into a new events file.
Does nothing if the EventFileWriter was not closed.

tf.compat.v1.summary.FileWriterCache

- **Contents**
- Class FileWriterCache
- Methods
  - clear
  - get

Class `FileWriterCache`

Cache for file writers.
Defined in `python/summary/writer/writer_cache.py`.
This class caches file writers, one per directory.

Methods

clear

```
@staticmethod
clear()
```

Clear cached summary writers. Currently only used for unit tests.

get

```
@staticmethod
get(logdir)
```

Returns the FileWriter for the specified directory.

Args:

- **`logdir`**: str, name of the directory.

Returns:

A `FileWriter`.

tf.compat.v1.summary.get_summary_description

Given a TensorSummary node_def, retrieve its SummaryDescription.

```
tf.compat.v1.summary.get_summary_description(node_def)
```

Defined in `python/summary/summary.py`.
When a Summary op is instantiated, a SummaryDescription of associated metadata is stored in its NodeDef. This method retrieves the description.

Args:

- **`node_def`**: the node_def_pb2.NodeDef of a TensorSummary op

Returns:

a summary_pb2.SummaryDescription

Raises:

- **`ValueError`**: if the node is not a summary op.

Eager Compatibility

Not compatible with eager execution. To write TensorBoard summaries under eager execution, use `tf.contrib.summary` instead.

tf.compat.v1.summary.histogram

Outputs a `Summary` protocol buffer with a histogram.

```
tf.compat.v1.summary.histogram(
    name,
    values,
    collections=None,
    family=None
)
```

Defined in `python/summary/summary.py`.

Adding a histogram summary makes it possible to visualize your data's distribution in TensorBoard. You can see a detailed explanation of the TensorBoard histogram dashboard here.

The generated `Summary` has one summary value containing a histogram for `values`.

This op reports an `InvalidArgument` error if any value is not finite.

Args:

- **`name`**: A name for the generated node. Will also serve as a series name in TensorBoard.
- **`values`**: A real numeric `Tensor`. Any shape. Values to use to build the histogram.
- **`collections`**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- **`family`**: Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

tf.compat.v1.summary.image

Outputs a `Summary` protocol buffer with images.

```
tf.compat.v1.summary.image(
    name,
    tensor,
    max_outputs=3,
    collections=None,
    family=None
)
```

Defined in `python/summary/summary.py`.

The summary has up to `max_outputs` summary values containing images. The images are built from `tensor` which must be 4-D with shape `[batch_size, height, width, channels]` and where `channels` can be:

- 1: `tensor` is interpreted as Grayscale.
- 3: `tensor` is interpreted as RGB.
- 4: `tensor` is interpreted as RGBA.

The images have the same number of channels as the input tensor. For float input, the values are normalized one image at a time to fit in the range `[0, 255]`. `uint8` values are unchanged. The op uses two different normalization algorithms:

- If the input values are all positive, they are rescaled so the largest one is 255.
- If any input value is negative, the values are shifted so input value 0.0 is at 127. They are then rescaled so that either the smallest value is 0, or the largest one is 255.

The `tag` in the outputted Summary.Value protobufs is generated based on the name, with a suffix depending on the max_outputs setting:

- If `max_outputs` is 1, the summary value tag is '*name*/image'.
- If `max_outputs` is greater than 1, the summary value tags are generated sequentially as '*name*/image/0', '*name*/image/1', etc.

Args:

- **`name`**: A name for the generated node. Will also serve as a series name in TensorBoard.
- **`tensor`**: A 4-D `uint8` or `float32` `Tensor` of shape `[batch_size, height, width, channels]` where `channels` is 1, 3, or 4.
- **`max_outputs`**: Max number of batch elements to generate images for.
- **`collections`**: Optional list of ops.GraphKeys. The collections to add the summary to. Defaults to [_ops.GraphKeys.SUMMARIES]
- **`family`**: Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

tf.compat.v1.summary.initialize

Initializes summary writing for graph execution mode.

```
tf.compat.v1.summary.initialize(
    graph=None,
    session=None
)
```

Defined in `python/ops/summary_ops_v2.py`.

This operation is a no-op when executing eagerly.

This helper method provides a higher-level alternative to using`tf.contrib.summary.summary_writer_initializer_op` and `tf.contrib.summary.graph`.

Most users will also want to call `tf.compat.v1.train.create_global_step` which can happen before or after this function is called.

Args:

- **`graph`**: A `tf.Graph` or `tf.compat.v1.GraphDef` to output to the writer. This function will not write the default graph by default. When writing to an event log file, the associated step will be zero.
- **`session`**: So this method can call `tf.Session.run`. This defaults to `tf.compat.v1.get_default_session`.

Raises:

- **`RuntimeError`**: If the current thread has no default `tf.contrib.summary.SummaryWriter`.
- **`ValueError`**: If session wasn't passed and no default session.

tf.compat.v1.summary.merge

Merges summaries.

```
tf.compat.v1.summary.merge(
    inputs,
```

```
    collections=None,
    name=None
)
```

Defined in `python/summary/summary.py`.

This op creates a `Summary` protocol buffer that contains the union of all the values in the input summaries.

When the Op is run, it reports an `InvalidArgument` error if multiple values in the summaries to merge use the same tag.

Args:

- **`inputs`**: A list of `string` `Tensor` objects containing serialized `Summary` protocol buffers.
- **`collections`**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[]`.
- **`name`**: A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer resulting from the merging.

Raises:

- **`RuntimeError`**: If called with eager mode enabled.

Eager Compatibility

Not compatible with eager execution. To write TensorBoard summaries under eager execution, use `tf.contrib.summary` instead.

tf.compat.v1.summary.merge_all

Merges all summaries collected in the default graph.

```
tf.compat.v1.summary.merge_all(
    key=tf.GraphKeys.SUMMARIES,
    scope=None,
    name=None
)
```

Defined in `python/summary/summary.py`.

Args:

- **`key`**: `GraphKey` used to collect the summaries. Defaults to `GraphKeys.SUMMARIES`.
- **`scope`**: Optional scope used to filter the summary ops, using `re.match`

Returns:

If no summaries were collected, returns None. Otherwise returns a scalar `Tensor` of type `string`containing the serialized `Summary` protocol buffer resulting from the merging.

Raises:

- **`RuntimeError`**: If called with eager execution enabled.

Eager Compatibility

Not compatible with eager execution. To write TensorBoard summaries under eager execution, use `tf.contrib.summary` instead.

tf.compat.v1.summary.scalar

Outputs a `Summary` protocol buffer containing a single scalar value.

```
tf.compat.v1.summary.scalar(
    name,
    tensor,
```

```
    collections=None,
    family=None
)
```

Defined in `python/summary/summary.py`.
The generated Summary has a Tensor.proto containing the input Tensor.

Args:

- **`name`**: A name for the generated node. Will also serve as the series name in TensorBoard.
- **`tensor`**: A real numeric Tensor containing a single value.
- **`collections`**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- **`family`**: Optional; if provided, used as the prefix of the summary tag name, which controls the tab name used for display on Tensorboard.

Returns:
A scalar `Tensor` of type `string`. Which contains a `Summary` protobuf.

Raises:

- **`ValueError`**: If tensor has the wrong shape or type.

tf.compat.v1.summary.SummaryDescription

- **Contents**
- Class SummaryDescription
- Properties
  - type_hint

Class `SummaryDescription`

Defined in `core/framework/summary.proto`.

Properties

`type_hint`

`string type_hint`

tf.compat.v1.summary.TaggedRunMetadata

- **Contents**
- Class TaggedRunMetadata
- Properties

Class `TaggedRunMetadata`

Defined in `core/util/event.proto`.

Properties

`run_metadata`

`bytes run_metadata`

`tag`

`string tag`

tf.compat.v1.summary.tensor_summary

Outputs a `Summary` protocol buffer with a serialized tensor.proto.

```
tf.compat.v1.summary.tensor_summary(
    name,
```

```
    tensor,
    summary_description=None,
    collections=None,
    summary_metadata=None,
    family=None,
    display_name=None
)
```

Defined in `python/summary/summary.py`.

Args:

- **`name`**: A name for the generated node. If display_name is not set, it will also serve as the tag name in TensorBoard. (In that case, the tag name will inherit tf name scopes.)
- **`tensor`**: A tensor of any type and shape to serialize.
- **`summary_description`**: A long description of the summary sequence. Markdown is supported.
- **`collections`**: Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- **`summary_metadata`**: Optional SummaryMetadata proto (which describes which plugins may use the summary value).
- **`family`**: Optional; if provided, used as the prefix of the summary tag, which controls the name used for display on TensorBoard when display_name is not set.
- **`display_name`**: A string used to name this data in TensorBoard. If this is not set, then the node name will be used instead.

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

tf.compat.v1.summary.text

Summarizes textual data.

```
tf.compat.v1.summary.text(
    name,
    tensor,
    collections=None
)
```

Defined in `python/summary/summary.py`.

Text data summarized via this plugin will be visible in the Text Dashboard in TensorBoard. The standard TensorBoard Text Dashboard will render markdown in the strings, and will automatically organize 1d and 2d tensors into tables. If a tensor with more than 2 dimensions is provided, a 2d subarray will be displayed along with a warning message. (Note that this behavior is not intrinsic to the text summary api, but rather to the default TensorBoard text plugin.)

Args:

- **`name`**: A name for the generated node. Will also serve as a series name in TensorBoard.
- **`tensor`**: a string-type Tensor to summarize.
- **`collections`**: Optional list of ops.GraphKeys. The collections to add the summary to. Defaults to [_ops.GraphKeys.SUMMARIES]

Returns:

A TensorSummary op that is configured so that TensorBoard will recognize that it contains textual data. The TensorSummary is a scalar `Tensor` of type `string` which contains `Summary` protobufs.

Raises:

- **`ValueError`**: If tensor has the wrong type.

Module: tf.sysconfig

- **Contents**
- Functions
- Other Members

System configuration library.

Functions

`get_compile_flags(...)`: Get the compilation flags for custom operators.
`get_include(...)`: Get the directory containing the TensorFlow C++ header files.
`get_lib(...)`: Get the directory containing the TensorFlow framework library.
`get_link_flags(...)`: Get the link flags for custom operators.

Other Members

- `CXX11_ABI_FLAG = 0`
- `MONOLITHIC_BUILD = 0`

tf.sysconfig.get_compile_flags

- **Contents**
- Aliases:

Get the compilation flags for custom operators.

Aliases:

- `tf.compat.v1.sysconfig.get_compile_flags`
- `tf.compat.v2.sysconfig.get_compile_flags`
- `tf.sysconfig.get_compile_flags`

```
tf.sysconfig.get_compile_flags()
```

Defined in `python/platform/sysconfig.py`.

Returns:
The compilation flags.

tf.sysconfig.get_include

- **Contents**
- Aliases:

Get the directory containing the TensorFlow C++ header files.

Aliases:

- `tf.compat.v1.sysconfig.get_include`
- `tf.compat.v2.sysconfig.get_include`
- `tf.sysconfig.get_include`

```
tf.sysconfig.get_include()
```

Defined in `python/platform/sysconfig.py`.

Returns:
The directory as string.

tf.sysconfig.get_lib

- **Contents**
- Aliases:

Get the directory containing the TensorFlow framework library.

Aliases:

- `tf.compat.v1.sysconfig.get_lib`
- `tf.compat.v2.sysconfig.get_lib`
- `tf.sysconfig.get_lib`

```
tf.sysconfig.get_lib()
```

Defined in `python/platform/sysconfig.py`.

Returns:
The directory as string.

tf.sysconfig.get_link_flags

- **Contents**
- Aliases:

Get the link flags for custom operators.

Aliases:

- `tf.compat.v1.sysconfig.get_link_flags`
- `tf.compat.v2.sysconfig.get_link_flags`
- `tf.sysconfig.get_link_flags`

```
tf.sysconfig.get_link_flags()
```

Defined in `python/platform/sysconfig.py`.

Returns:
The link flags.

Module: tf.compat.v1.test / tf.test

- **Contents**
- Classes
- Functions

Testing.
See the Testing guide.
Note: `tf.compat.v1.test.mock` is an alias to the python `mock` or `unittest.mock` depending on the python version.

Classes

`class Benchmark`: Abstract class that provides helpers for TensorFlow benchmarks.
`class StubOutForTesting`: Support class for stubbing methods out for unit testing.
`class TestCase`: Base class for tests that need to test TensorFlow.

Functions

`assert_equal_graph_def(...)`: Asserts that two `GraphDef`s are (mostly) the same.
`benchmark_config(...)`: Returns a tf.compat.v1.ConfigProto for disabling the dependency optimizer.
`compute_gradient(...)`: Computes and returns the theoretical and numerical Jacobian.
`compute_gradient_error(...)`: Computes the gradient error.
`create_local_cluster(...)`: Create and start local servers and return the associated `Server`objects.
`get_temp_dir(...)`: Returns a temporary directory for use during tests.
`gpu_device_name(...)`: Returns the name of a GPU device if available or the empty string.
`is_built_with_cuda(...)`: Returns whether TensorFlow was built with CUDA (GPU) support.

`is_gpu_available(...)`: Returns whether TensorFlow can access a GPU.
`main(...)`: Runs all unit tests.
`test_src_dir_path(...)`: Creates an absolute test srcdir path given a relative path.

tf.compat.v1.test.assert_equal_graph_def

Asserts that two `GraphDef`s are (mostly) the same.

```
tf.compat.v1.test.assert_equal_graph_def(
    actual,
    expected,
    checkpoint_v2=False,
    hash_table_shared_name=False
)
```

Defined in `python/framework/test_util.py`.
Compares two `GraphDef` protos for equality, ignoring versions and ordering of nodes, attrs, and control inputs. Node names are used to match up nodes between the graphs, so the naming of nodes must be consistent.

Args:

- **`actual`**: The `GraphDef` we have.
- **`expected`**: The `GraphDef` we expected.
- **`checkpoint_v2`**: boolean determining whether to ignore randomized attribute values that appear in V2 checkpoints.
- **`hash_table_shared_name`**: boolean determining whether to ignore randomized shared_names that appear in HashTableV2 op defs.

Raises:

- **`AssertionError`**: If the `GraphDef`s do not match.
- **`TypeError`**: If either argument is not a `GraphDef`.

tf.compat.v1.test.compute_gradient

Computes and returns the theoretical and numerical Jacobian.

```
tf.compat.v1.test.compute_gradient(
    x,
    x_shape,
    y,
    y_shape,
    x_init_value=None,
    delta=0.001,
    init_targets=None,
    extra_feed_dict=None
)
```

Defined in `python/ops/gradient_checker.py`.
If `x` or `y` is complex, the Jacobian will still be real but the corresponding Jacobian dimension(s) will be twice as large. This is required even if both input and output is complex since TensorFlow graphs are not necessarily holomorphic, and may have gradients not expressible as complex numbers. For example, if `x` is complex with shape `[m]` and `y` is complex with shape `[n]`, each Jacobian `J` will have shape `[m * 2, n * 2]` with

```
J[:m, :n] = d(Re y)/d(Re x)
J[:m, n:] = d(Im y)/d(Re x)
```

```
J[m:, :n] = d(Re y)/d(Im x)
J[m:, n:] = d(Im y)/d(Im x)
```

Args:

- `x`: a tensor or list of tensors
- `x_shape`: the dimensions of x as a tuple or an array of ints. If x is a list, then this is the list of shapes.
- `y`: a tensor
- `y_shape`: the dimensions of y as a tuple or an array of ints.
- `x_init_value`: (optional) a numpy array of the same shape as "x" representing the initial value of x. If x is a list, this should be a list of numpy arrays. If this is none, the function will pick a random tensor as the initial value.
- `delta`: (optional) the amount of perturbation.
- `init_targets`: list of targets to run to initialize model params.
- `extra_feed_dict`: dict that allows fixing specified tensor values during the Jacobian calculation.

Returns:
Two 2-d numpy arrays representing the theoretical and numerical Jacobian for dy/dx. Each has "x_size" rows and "y_size" columns where "x_size" is the number of elements in x and "y_size" is the number of elements in y. If x is a list, returns a list of two numpy arrays.

tf.compat.v1.test.compute_gradient_error

Computes the gradient error.

```
tf.compat.v1.test.compute_gradient_error(
    x,
    x_shape,
    y,
    y_shape,
    x_init_value=None,
    delta=0.001,
    init_targets=None,
    extra_feed_dict=None
)
```

Defined in `python/ops/gradient_checker.py`.
Computes the maximum error for dy/dx between the computed Jacobian and the numerically estimated Jacobian.
This function will modify the tensors passed in as it adds more operations and hence changing the consumers of the operations of the input tensors.
This function adds operations to the current session. To compute the error using a particular device, such as a GPU, use the standard methods for setting a device (e.g. using with sess.graph.device() or setting a device function in the session constructor).

Args:

- `x`: a tensor or list of tensors
- `x_shape`: the dimensions of x as a tuple or an array of ints. If x is a list, then this is the list of shapes.
- `y`: a tensor
- `y_shape`: the dimensions of y as a tuple or an array of ints.
- `x_init_value`: (optional) a numpy array of the same shape as "x" representing the initial value of x. If x is a list, this should be a list of numpy arrays. If this is none, the function will pick a random tensor as the initial value.
- `delta`: (optional) the amount of perturbation.

- `init_targets`: list of targets to run to initialize model params.
- `extra_feed_dict`: dict that allows fixing specified tensor values during the Jacobian calculation.

Returns:
The maximum error in between the two Jacobians.

tf.compat.v1.test.get_temp_dir

Returns a temporary directory for use during tests.

```
tf.compat.v1.test.get_temp_dir()
```

Defined in `python/platform/test.py`.
There is no need to delete the directory after the test.

Returns:
The temporary directory.

tf.compat.v1.test.StubOutForTesting

- Contents
- Class StubOutForTesting
- __init__
- Methods
  - CleanUp

Class `StubOutForTesting`

Support class for stubbing methods out for unit testing.
Defined in `python/platform/googletest.py`.

Sample Usage:
You want os.path.exists() to always return true during testing.
stubs = StubOutForTesting() stubs.Set(os.path, 'exists', lambda x: 1) ... stubs.CleanUp()
The above changes os.path.exists into a lambda that returns 1. Once the ... part of the code finishes, the CleanUp() looks up the old value of os.path.exists and restores it.

`__init__`

```
__init__()
```

Methods

`CleanUp`

```
CleanUp()
```

Undoes all SmartSet() & Set() calls, restoring original definitions.

`Set`

```
Set(
    parent,
    child_name,
    new_child
)
```

In parent, replace child_name's old definition with new_child.

The parent could be a module when the child is a function at module scope. Or the parent could be a class when a class' method is being replaced. The named child is set to new_child, while the prior definition is saved away for later, when UnsetAll() is called.
This method supports the case where child_name is a staticmethod or a classmethod of parent.

Args:

- `parent`: The context in which the attribute child_name is to be changed.
- `child_name`: The name of the attribute to change.
- `new_child`: The new value of the attribute.

SmartSet

```
SmartSet(
    obj,
    attr_name,
    new_attr
)
```

Replace obj.attr_name with new_attr.
This method is smart and works at the module, class, and instance level while preserving proper inheritance. It will not stub out C types however unless that has been explicitly allowed by the type.
This method supports the case where attr_name is a staticmethod or a classmethod of obj.

Notes:

- If obj is an instance, then it is its class that will actually be stubbed. Note that the method Set() does not do that: if obj is an instance, it (and not its class) will be stubbed.
- The stubbing is using the builtin getattr and setattr. So, the **get** and **set** will be called when stubbing (TODO: A better idea would probably be to manipulate obj.**dict** instead of getattr() and setattr()).

Args:

- `obj`: The object whose attributes we want to modify.
- `attr_name`: The name of the attribute to modify.
- `new_attr`: The new value for the attribute.

Raises:

- `AttributeError`: If the attribute cannot be found.

SmartUnsetAll

```
SmartUnsetAll()
```

Reverses SmartSet() calls, restoring things to original definitions.
This method is automatically called when the StubOutForTesting() object is deleted; there is no need to call it explicitly.
It is okay to call SmartUnsetAll() repeatedly, as later calls have no effect if no SmartSet() calls have been made.

UnsetAll

```
UnsetAll()
```

Reverses Set() calls, restoring things to their original definitions.
This method is automatically called when the StubOutForTesting() object is deleted; there is no need to call it explicitly.
It is okay to call UnsetAll() repeatedly, as later calls have no effect if no Set() calls have been made.

__enter__

```
__enter__()
```

__exit__

```
__exit__(
    unused_exc_type,
    unused_exc_value,
    unused_tb
)
```

tf.compat.v1.test.test_src_dir_path

Creates an absolute test srcdir path given a relative path.

```
tf.compat.v1.test.test_src_dir_path(relative_path)
```

Defined in `python/platform/test.py`.

Args:

- **`relative_path`**: a path relative to tensorflow root. e.g. "core/platform".

Returns:

An absolute path to the linked in runfiles.

tf.test.assert_equal_graph_def

- Contents
- Aliases:

Asserts that two `GraphDef`s are (mostly) the same.

Aliases:

- `tf.compat.v2.test.assert_equal_graph_def`
- `tf.test.assert_equal_graph_def`

```
tf.test.assert_equal_graph_def(
    expected,
    actual
)
```

Defined in `python/framework/test_util.py`.

Compares two `GraphDef` protos for equality, ignoring versions and ordering of nodes, attrs, and control inputs. Node names are used to match up nodes between the graphs, so the naming of nodes must be consistent. This function ignores randomized attribute values that may appear in V2 checkpoints.

Args:

- **`expected`**: The `GraphDef` we expected.
- **`actual`**: The `GraphDef` we have.

Raises:

- **`AssertionError`**: If the `GraphDef`s do not match.
- **`TypeError`**: If either argument is not a `GraphDef`.

tf.test.Benchmark

- Contents

- Class Benchmark
  - Aliases:
- __init__
- Methods

Class `Benchmark`

Abstract class that provides helpers for TensorFlow benchmarks.

Aliases:

- Class `tf.compat.v1.test.Benchmark`
- Class `tf.compat.v2.test.Benchmark`
- Class `tf.test.Benchmark`

Defined in `python/platform/benchmark.py`.

`__init__`

```
__init__()
```

Methods

`evaluate`

```
evaluate(tensors)
```

Evaluates tensors and returns numpy values.

Args:

- **`tensors`**: A Tensor or a nested list/tuple of Tensors.

Returns:

tensors numpy values.

`is_abstract`

```
@classmethod
is_abstract(cls)
```

`report_benchmark`

```
report_benchmark(
    iters=None,
    cpu_time=None,
    wall_time=None,
    throughput=None,
    extras=None,
    name=None,
    metrics=None
)
```

Report a benchmark.

Args:

- **`iters`**: (optional) How many iterations were run
- **`cpu_time`**: (optional) Median or mean cpu time in seconds.
- **`wall_time`**: (optional) Median or mean wall time in seconds.
- **`throughput`**: (optional) Throughput (in MB/s)

- **`extras`**: (optional) Dict mapping string keys to additional benchmark info. Values may be either floats or values that are convertible to strings.
- **`name`**: (optional) Override the BenchmarkEntry name with `name`. Otherwise it is inferred from the top-level method name.
- **`metrics`**: (optional) A list of dict, where each dict has the keys below name (required), string, metric name value (required), double, metric value min_value (optional), double, minimum acceptable metric value max_value (optional), double, maximum acceptable metric value

run_op_benchmark

```
run_op_benchmark(
    sess,
    op_or_tensor,
    feed_dict=None,
    burn_iters=2,
    min_iters=10,
    store_trace=False,
    store_memory_usage=True,
    name=None,
    extras=None,
    mbs=0
)
```

Run an op or tensor in the given session. Report the results.

Args:

- **`sess`**: `Session` object to use for timing.
- **`op_or_tensor`**: `Operation` or `Tensor` to benchmark.
- **`feed_dict`**: A `dict` of values to feed for each op iteration (see the `feed_dict` parameter of `Session.run`).
- **`burn_iters`**: Number of burn-in iterations to run.
- **`min_iters`**: Minimum number of iterations to use for timing.
- **`store_trace`**: Boolean, whether to run an extra untimed iteration and store the trace of iteration in returned extras. The trace will be stored as a string in Google Chrome trace format in the extras field "full_trace_chrome_format". Note that trace will not be stored in test_log_pb2.TestResults proto.
- **`store_memory_usage`**: Boolean, whether to run an extra untimed iteration, calculate memory usage, and store that in extras fields.
- **`name`**: (optional) Override the BenchmarkEntry name with `name`. Otherwise it is inferred from the top-level method name.
- **`extras`**: (optional) Dict mapping string keys to additional benchmark info. Values may be either floats or values that are convertible to strings.
- **`mbs`**: (optional) The number of megabytes moved by this op, used to calculate the ops throughput.

Returns:

A `dict` containing the key-value pairs that were passed to `report_benchmark`. If `store_trace`option is used, then `full_chrome_trace_format` will be included in return dictionary even though it is not passed to `report_benchmark` with `extras`.

tf.test.benchmark_config

- Contents
- Aliases:

Returns a tf.compat.v1.ConfigProto for disabling the dependency optimizer.

Aliases:

- `tf.compat.v1.test.benchmark_config`
- `tf.compat.v2.test.benchmark_config`
- `tf.test.benchmark_config`

```
tf.test.benchmark_config()
```

Defined in `python/platform/benchmark.py`.

Returns:

A TensorFlow ConfigProto object.

tf.test.compute_gradient

- Contents
- Aliases:

Computes the theoretical and numeric Jacobian of `f`.

Aliases:

- `tf.compat.v2.test.compute_gradient`
- `tf.test.compute_gradient`

```
tf.test.compute_gradient(
    f,
    x,
    delta=0.001
)
```

Defined in `python/ops/gradient_checker_v2.py`.

With y = f(x), computes the theoretical and numeric Jacobian dy/dx.

Args:

- **`f`**: the function.
- **`x`**: a list arguments for the function
- **`delta`**: (optional) perturbation used to compute numeric Jacobian.

Returns:

A pair of lists, where the first is a list of 2-d numpy arrays representing the theoretical Jacobians for each argument, and the second list is the numerical ones. Each 2-d array has "x_size" rows and "y_size" columns where "x_size" is the number of elements in the corresponding argument and "y_size" is the number of elements in f(x).

Raises:

- **`ValueError`**: If result is empty but the gradient is nonzero.
- **`ValueError`**: If x is not list, but any other type.

Example:

```
@tf.function
def test_func(x):
  return x*x

theoretical, numerical = tf.test.compute_gradient(test_func, [1.0])
theoretical, numerical
# ((array([[2.]], dtype=float32),), (array([[2.000004]], dtype=float32),))
```

tf.test.create_local_cluster

- Contents
- Aliases:

Create and start local servers and return the associated `Server` objects.

Aliases:

- `tf.compat.v1.test.create_local_cluster`
- `tf.compat.v2.test.create_local_cluster`
- `tf.test.create_local_cluster`

```
tf.test.create_local_cluster(
    num_workers,
    num_ps,
    protocol='grpc',
    worker_config=None,
    ps_config=None
)
```

Defined in `python/framework/test_util.py`.

"PS" stands for "parameter server": a task responsible for storing and updating the model's parameters. Other tasks send updates to these parameters as they work on optimizing the parameters. This particular division of labor between tasks is not required, but is common for distributed training.

Read more at https://www.tensorflow.org/guide/extend/architecture

Figure illustrates the interaction of these components. "/job:worker/task:0" and "/job:ps/task:0" are both tasks with worker services.

Example:

```
workers, _ = tf.test.create_local_cluster(num_workers=2, num_ps=2)

worker_sessions = [tf.compat.v1.Session(w.target) for w in workers]

with tf.device("/job:ps/task:0"):
  ...
with tf.device("/job:ps/task:1"):
  ...
with tf.device("/job:worker/task:0"):
  ...
with tf.device("/job:worker/task:1"):
  ...

worker_sessions[0].run(...)
```

Args:

- **`num_workers`**: Number of worker servers to start.
- **`num_ps`**: Number of PS servers to start.
- **`protocol`**: Communication protocol. Allowed values are documented in the documentation of `tf.distribute.Server`.
- **`worker_config`**: (optional) `tf.ConfigProto` to initialize workers. Can be used to instantiate multiple devices etc.

- **`ps_config`**: (optional) `tf.ConfigProto` to initialize PS servers.

  Returns:
  A tuple `(worker_servers, ps_servers)`. `worker_servers` is a list of `num_workers` objects of type `tf.distribute.Server` (all running locally); and `ps_servers` is a list of `num_ps` objects of similar type.

  Raises:
- **`ImportError`**: if portpicker module was not found at load time

tf.test.gpu_device_name

- Contents
- Aliases:

Returns the name of a GPU device if available or the empty string.

Aliases:

- `tf.compat.v1.test.gpu_device_name`
- `tf.compat.v2.test.gpu_device_name`
- `tf.test.gpu_device_name`

```
tf.test.gpu_device_name()
```

Defined in `python/framework/test_util.py`.

tf.test.is_built_with_cuda

- Contents
- Aliases:

Returns whether TensorFlow was built with CUDA (GPU) support.

Aliases:

- `tf.compat.v1.test.is_built_with_cuda`
- `tf.compat.v2.test.is_built_with_cuda`
- `tf.test.is_built_with_cuda`

```
tf.test.is_built_with_cuda()
```

Defined in `python/platform/test.py`.

tf.test.is_gpu_available

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Returns whether TensorFlow can access a GPU.

Aliases:

- `tf.compat.v1.test.is_gpu_available`
- `tf.compat.v2.test.is_gpu_available`
- `tf.test.is_gpu_available`

```
tf.test.is_gpu_available(
    cuda_only=False,
    min_cuda_compute_capability=None
)
```

Defined in `python/framework/test_util.py`.

Used in the guide:

- [Eager essentials]()

Used in the tutorials:

- [Tensors and Operations]()
- [Text classification of movie reviews with Keras and TensorFlow Hub]()

Warning: if a non-GPU version of the package is installed, the function would also return False. Use `tf.test.is_built_with_cuda` to validate if TensorFlow was build with CUDA support.

Args:

- `cuda_only`: limit the search to CUDA GPUs.
- `min_cuda_compute_capability`: a (major,minor) pair that indicates the minimum CUDA compute capability required, or None if no requirement.

Returns:
True if a GPU device of the requested kind is available.

tf.test.main

- Contents
- Aliases:

Runs all unit tests.

Aliases:

- `tf.compat.v1.test.main`
- `tf.compat.v2.test.main`
- `tf.test.main`

```
tf.test.main(argv=None)
```

Defined in `python/platform/test.py`.

tf.test.TestCase

- Contents
- Class TestCase
  - Aliases:
- __init__
- Child Classes

Class `TestCase`

Base class for tests that need to test TensorFlow.

Aliases:

- Class `tf.compat.v1.test.TestCase`
- Class `tf.compat.v2.test.TestCase`
- Class `tf.test.TestCase`

Defined in `python/framework/test_util.py`.

`__init__`

```
__init__(methodName='runTest')
```

Child Classes

`class failureException`

Methods

__call__

```
__call__(
    *args,
    **kwds
)
```

__eq__

```
__eq__(other)
```

addCleanup

```
addCleanup(
    function,
    *args,
    **kwargs
)
```

Add a function, with arguments, to be called when the test is completed. Functions added are called on a LIFO basis and are called after tearDown on test failure or success.
Cleanup items are called even if setUp fails (unlike tearDown).

addTypeEqualityFunc

```
addTypeEqualityFunc(
    typeobj,
    function
)
```

Add a type specific assertEqual style function to compare a type.
This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Args:

- **`typeobj`**: The data type to call this function on when both values are of the same type in assertEqual().
- **`function`**: The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

assertAllClose

```
assertAllClose(
    a,
    b,
    rtol=1e-06,
    atol=1e-06,
    msg=None
)
```

Asserts that two structures of numpy arrays or Tensors, have near values.

`a` and `b` can be arbitrarily nested structures. A layer of a nested structure can be a `dict`, `namedtuple`, `tuple` or `list`.

Args:

- **`a`**: The expected numpy `ndarray`, or anything that can be converted into a numpy `ndarray`(including Tensor), or any arbitrarily nested of structure of these.
- **`b`**: The actual numpy `ndarray`, or anything that can be converted into a numpy `ndarray`(including Tensor), or any arbitrarily nested of structure of these.
- **`rtol`**: relative tolerance.
- **`atol`**: absolute tolerance.
- **`msg`**: Optional message to report on failure.

Raises:

- **`ValueError`**: if only one of `a[p]` and `b[p]` is a dict or `a[p]` and `b[p]` have different length, where `[p]` denotes a path to the nested structure, e.g. given `a = [(1, 1), {'d': (6, 7)}]`and `[p] = [1]['d']`, then `a[p] = (6, 7)`.

`assertAllCloseAccordingToType`

```
assertAllCloseAccordingToType(
    a,
    b,
    rtol=1e-06,
    atol=1e-06,
    float_rtol=1e-06,
    float_atol=1e-06,
    half_rtol=0.001,
    half_atol=0.001,
    bfloat16_rtol=0.01,
    bfloat16_atol=0.01,
    msg=None
)
```

Like assertAllClose, but also suitable for comparing fp16 arrays.
In particular, the tolerance is reduced to 1e-3 if at least one of the arguments is of type float16.

Args:

- **`a`**: the expected numpy ndarray or anything can be converted to one.
- **`b`**: the actual numpy ndarray or anything can be converted to one.
- **`rtol`**: relative tolerance.
- **`atol`**: absolute tolerance.
- **`float_rtol`**: relative tolerance for float32.
- **`float_atol`**: absolute tolerance for float32.
- **`half_rtol`**: relative tolerance for float16.
- **`half_atol`**: absolute tolerance for float16.
- **`bfloat16_rtol`**: relative tolerance for bfloat16.
- **`bfloat16_atol`**: absolute tolerance for bfloat16.
- **`msg`**: Optional message to report on failure.

`assertAllEqual`

```
assertAllEqual(
    a,
    b,
```

```
    msg=None
)
```

Asserts that two numpy arrays or Tensors have the same values.

Args:

- `a`: the expected numpy ndarray or anything can be converted to one.
- `b`: the actual numpy ndarray or anything can be converted to one.
- `msg`: Optional message to report on failure.

assertAllGreater

```
assertAllGreater(
    a,
    comparison_target
)
```

Assert element values are all greater than a target value.

Args:

- `a`: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray` (including Tensor).
- `comparison_target`: The target value of comparison.

assertAllGreaterEqual

```
assertAllGreaterEqual(
    a,
    comparison_target
)
```

Assert element values are all greater than or equal to a target value.

Args:

- `a`: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray` (including Tensor).
- `comparison_target`: The target value of comparison.

assertAllInRange

```
assertAllInRange(
    target,
    lower_bound,
    upper_bound,
    open_lower_bound=False,
    open_upper_bound=False
)
```

Assert that elements in a Tensor are all in a given range.

Args:

- `target`: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray`(including Tensor).
- `lower_bound`: lower bound of the range
- `upper_bound`: upper bound of the range
- `open_lower_bound`: (`bool`) whether the lower bound is open (i.e., > rather than the default >=)
- `open_upper_bound`: (`bool`) whether the upper bound is open (i.e., < rather than the default <=)

Raises:

- **`AssertionError`**: if the value tensor does not have an ordered numeric type (float* or int*), or if there are nan values, or if any of the elements do not fall in the specified range.

`assertAllInSet`

```
assertAllInSet(
    target,
    expected_set
)
```

Assert that elements of a Tensor are all in a given closed set.

Args:

- **`target`**: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray`(including Tensor).
- **`expected_set`**: (`list`, `tuple` or `set`) The closed set that the elements of the value of `target` are expected to fall into.

Raises:

- **`AssertionError`**: if any of the elements do not fall into `expected_set`.

`assertAllLess`

```
assertAllLess(
    a,
    comparison_target
)
```

Assert element values are all less than a target value.

Args:

- **`a`**: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray` (including Tensor).
- **`comparison_target`**: The target value of comparison.

`assertAllLessEqual`

```
assertAllLessEqual(
    a,
    comparison_target
)
```

Assert element values are all less than or equal to a target value.

Args:

- **`a`**: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray` (including Tensor).
- **`comparison_target`**: The target value of comparison.

`assertAlmostEqual`

```
assertAlmostEqual(
    first,
    second,
    places=None,
    msg=None,
    delta=None
```

```
)
```

Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).
If the two objects compare equal then they will automatically compare almost equal.

assertAlmostEquals

```
assertAlmostEquals(
    *args,
    **kwargs
)
```

assertArrayNear

```
assertArrayNear(
    farray1,
    farray2,
    err,
    msg=None
)
```

Asserts that two float arrays are near each other.
Checks that for all elements of farray1 and farray2 |f1 - f2| < err. Asserts a test failure if not.

Args:

- **`farray1`**: a list of float values.
- **`farray2`**: a list of float values.
- **`err`**: a float value.
- **`msg`**: Optional message to report on failure.

assertBetween

```
assertBetween(
    value,
    minv,
    maxv,
    msg=None
)
```

Asserts that value is between minv and maxv (inclusive).

assertCommandFails

```
assertCommandFails(
    command,
    regexes,
    env=None,
    close_fds=True,
    msg=None
```

```
)
```

Asserts a shell command fails and the error matches a regex in a list.

Args:

- **`command`**: List or string representing the command to run.
- **`regexes`**: the list of regular expression strings.
- **`env`**: Dictionary of environment variable settings. If None, no environment variables will be set for the child process. This is to make tests more hermetic. NOTE: this behavior is different than the standard subprocess module.
- **`close_fds`**: Whether or not to close all open fd's in the child after forking.
- **`msg`**: Optional message to report on failure.

assertCommandSucceeds

```
assertCommandSucceeds(
    command,
    regexes=(b'',),
    env=None,
    close_fds=True,
    msg=None
)
```

Asserts that a shell command succeeds (i.e. exits with code 0).

Args:

- **`command`**: List or string representing the command to run.
- **`regexes`**: List of regular expression byte strings that match success.
- **`env`**: Dictionary of environment variable settings. If None, no environment variables will be set for the child process. This is to make tests more hermetic. NOTE: this behavior is different than the standard subprocess module.
- **`close_fds`**: Whether or not to close all open fd's in the child after forking.
- **`msg`**: Optional message to report on failure.

assertContainsExactSubsequence

```
assertContainsExactSubsequence(
    container,
    subsequence,
    msg=None
)
```

Asserts that "container" contains "subsequence" as an exact subsequence.
Asserts that "container" contains all the elements of "subsequence", in order, and without other elements interspersed. For example, [1, 2, 3] is an exact subsequence of [0, 0, 1, 2, 3, 0] but not of [0, 0, 1, 2, 0, 3, 0].

Args:

- **`container`**: the list we're testing for subsequence inclusion.
- **`subsequence`**: the list we hope will be an exact subsequence of container.
- **`msg`**: Optional message to report on failure.

assertContainsInOrder

```
assertContainsInOrder(
    strings,
```

```
    target,
    msg=None
)
```

Asserts that the strings provided are found in the target in order.
This may be useful for checking HTML output.

Args:

- **`strings`**: A list of strings, such as ['fox', 'dog']
- **`target`**: A target string in which to look for the strings, such as 'The quick brown fox jumped over the lazy dog'.
- **`msg`**: Optional message to report on failure.

assertContainsSubsequence

```
assertContainsSubsequence(
    container,
    subsequence,
    msg=None
)
```

Asserts that "container" contains "subsequence" as a subsequence.
Asserts that "container" contains all the elements of "subsequence", in order, but possibly with other elements interspersed. For example, [1, 2, 3] is a subsequence of [0, 0, 1, 2, 0, 3, 0] but not of [0, 0, 1, 3, 0, 2, 0].

Args:

- **`container`**: the list we're testing for subsequence inclusion.
- **`subsequence`**: the list we hope will be a subsequence of container.
- **`msg`**: Optional message to report on failure.

assertContainsSubset

```
assertContainsSubset(
    expected_subset,
    actual_set,
    msg=None
)
```

Checks whether actual iterable is a superset of expected iterable.

assertCountEqual

```
assertCountEqual(
    first,
    second,
    msg=None
)
```

An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

```
self.assertEqual(Counter(list(first)),
                 Counter(list(second)))
```

Example: - [0, 1, 1] and [1, 0, 1] compare equal. - [0, 0, 1] and [0, 1] compare unequal.

assertDTypeEqual

```
assertDTypeEqual(
    target,
    expected_dtype
)
```

Assert ndarray data type is equal to expected.

Args:

- **`target`**: The numpy `ndarray`, or anything that can be converted into a numpy `ndarray`(including Tensor).
- **`expected_dtype`**: Expected data type.

assertDeviceEqual

```
assertDeviceEqual(
    device1,
    device2,
    msg=None
)
```

Asserts that the two given devices are the same.

Args:

- **`device1`**: A string device name or TensorFlow `DeviceSpec` object.
- **`device2`**: A string device name or TensorFlow `DeviceSpec` object.
- **`msg`**: Optional message to report on failure.

assertDictContainsSubset

```
assertDictContainsSubset(
    subset,
    dictionary,
    msg=None
)
```

Checks whether dictionary is a superset of subset.

assertDictEqual

```
assertDictEqual(
    a,
    b,
    msg=None
)
```

Raises AssertionError if a and b are not equal dictionaries.

Args:

- **`a`**: A dict, the expected value.
- **`b`**: A dict, the actual value.
- **`msg`**: An optional str, the associated message.

Raises:

- **`AssertionError`**: if the dictionaries are not equal.

`assertEmpty`

```
assertEmpty(
    container,
    msg=None
)
```

Asserts that an object has zero length.

Args:

- **`container`**: Anything that implements the collections.Sized interface.
- **`msg`**: Optional message to report on failure.

`assertEndsWith`

```
assertEndsWith(
    actual,
    expected_end,
    msg=None
)
```

Asserts that actual.endswith(expected_end) is True.

Args:

- **`actual`**: str
- **`expected_end`**: str
- **`msg`**: Optional message to report on failure.

`assertEqual`

```
assertEqual(
    first,
    second,
    msg=None
)
```

Fail if the two objects are unequal as determined by the '==' operator.

`assertEquals`

```
assertEquals(
    *args,
    **kwargs
)
```

`assertFalse`

```
assertFalse(
    expr,
    msg=None
)
```

Check that the expression is false.

assertGreater

```
assertGreater(
    a,
    b,
    msg=None
)
```

Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual

```
assertGreaterEqual(
    a,
    b,
    msg=None
)
```

Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn

```
assertIn(
    member,
    container,
    msg=None
)
```

Just like self.assertTrue(a in b), but with a nicer default message.

assertIs

```
assertIs(
    expr1,
    expr2,
    msg=None
)
```

Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance

```
assertIsInstance(
    obj,
    cls,
    msg=None
)
```

Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone

```
assertIsNone(
    obj,
    msg=None
```

```
)
```

Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot

```
assertIsNot(
    expr1,
    expr2,
    msg=None
)
```

Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone

```
assertIsNotNone(
    obj,
    msg=None
)
```

Included for symmetry with assertIsNone.

assertItemsEqual

```
assertItemsEqual(
    first,
    second,
    msg=None
)
```

An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

```
self.assertEqual(Counter(list(first)),
                 Counter(list(second)))
```

Example: - [0, 1, 1] and [1, 0, 1] compare equal. - [0, 0, 1] and [0, 1] compare unequal.

assertJsonEqual

```
assertJsonEqual(
    first,
    second,
    msg=None
)
```

Asserts that the JSON objects defined in two strings are equal.
A summary of the differences will be included in the failure message using assertSameStructure.

Args:

- **`first`**: A string contining JSON to decode and compare to second.
- **`second`**: A string contining JSON to decode and compare to first.
- **`msg`**: Additional text to include in the failure message.

assertLen

```
assertLen(
    container,
    expected_len,
    msg=None
)
```

Asserts that an object has the expected length.

Args:

- **`container`**: Anything that implements the collections.Sized interface.
- **`expected_len`**: The expected length of the container.
- **`msg`**: Optional message to report on failure.

assertLess

```
assertLess(
    a,
    b,
    msg=None
)
```

Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual

```
assertLessEqual(
    a,
    b,
    msg=None
)
```

Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual

```
assertListEqual(
    list1,
    list2,
    msg=None
)
```

A list-specific equality assertion.

Args:

- **`list1`**: The first list to compare.
- **`list2`**: The second list to compare.
- **`msg`**: Optional message to use on failure instead of a list of differences.

assertLogs

```
assertLogs(
    logger=None,
    level=None
```

```
)
```

Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.
This method must be used as a context manager, and will yield a recording object with two attributes: `output` and `records`. At the end of the context manager, the `output` attribute will be a list of the matching formatted log messages and the `records` attribute will be a list of the corresponding LogRecord objects.
Example::

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual

```
assertMultiLineEqual(
    first,
    second,
    msg=None,
    **kwargs
)
```

Asserts that two multi-line strings are equal.

assertNDArrayNear

```
assertNDArrayNear(
    ndarray1,
    ndarray2,
    err,
    msg=None
)
```

Asserts that two numpy arrays have near values.

Args:

- **`ndarray1`**: a numpy ndarray.
- **`ndarray2`**: a numpy ndarray.
- **`err`**: a float. The maximum absolute difference allowed.
- **`msg`**: Optional message to report on failure.

assertNear

```
assertNear(
    f1,
    f2,
    err,
    msg=None
)
```

Asserts that two floats are near each other.

Checks that |f1 - f2| < err and asserts a test failure if not.

Args:

- **`f1`**: A float value.
- **`f2`**: A float value.
- **`err`**: A float value.
- **`msg`**: An optional string message to append to the failure message.

assertNoCommonElements

```
assertNoCommonElements(
    expected_seq,
    actual_seq,
    msg=None
)
```

Checks whether actual iterable and expected iterable are disjoint.

assertNotAllClose

```
assertNotAllClose(
    a,
    b,
    **kwargs
)
```

Assert that two numpy arrays, or Tensors, do not have near values.

Args:

- **`a`**: the first value to compare.
- **`b`**: the second value to compare.
- **`**kwargs`**: additional keyword arguments to be passed to the underlying `assertAllClose` call.

Raises:

- **`AssertionError`**: If `a` and `b` are unexpectedly close at all elements.

assertNotAlmostEqual

```
assertNotAlmostEqual(
    first,
    second,
    places=None,
    msg=None,
    delta=None
)
```

Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotAlmostEquals

```
assertNotAlmostEquals(
    *args,
```

```
    **kwargs
)
```

assertNotEmpty

```
assertNotEmpty(
    container,
    msg=None
)
```

Asserts that an object has non-zero length.

Args:

- **`container`**: Anything that implements the collections.Sized interface.
- **`msg`**: Optional message to report on failure.

assertNotEndsWith

```
assertNotEndsWith(
    actual,
    unexpected_end,
    msg=None
)
```

Asserts that actual.endswith(unexpected_end) is False.

Args:

- **`actual`**: str
- **`unexpected_end`**: str
- **`msg`**: Optional message to report on failure.

assertNotEqual

```
assertNotEqual(
    first,
    second,
    msg=None
)
```

Fail if the two objects are equal as determined by the '!=' operator.

assertNotEquals

```
assertNotEquals(
    *args,
    **kwargs
)
```

assertNotIn

```
assertNotIn(
    member,
    container,
    msg=None
```

```
)
```

Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance

```
assertNotIsInstance(
    obj,
    cls,
    msg=None
)
```

Included for symmetry with assertIsInstance.

assertNotRegex

```
assertNotRegex(
    text,
    unexpected_regex,
    msg=None
)
```

Fail the test if the text matches the regular expression.

assertNotStartsWith

```
assertNotStartsWith(
    actual,
    unexpected_start,
    msg=None
)
```

Asserts that actual.startswith(unexpected_start) is False.

Args:

- **`actual`**: str
- **`unexpected_start`**: str
- **`msg`**: Optional message to report on failure.

assertProtoEquals

```
assertProtoEquals(
    expected_message_maybe_ascii,
    message,
    msg=None
)
```

Asserts that message is same as parsed expected_message_ascii.
Creates another prototype of message, reads the ascii message into it and then compares them using self._AssertProtoEqual().

Args:

- **`expected_message_maybe_ascii`**: proto message in original or ascii form.
- **`message`**: the message to validate.
- **`msg`**: Optional message to report on failure.

assertProtoEqualsVersion

```
assertProtoEqualsVersion(
    expected,
    actual,
    producer=versions.GRAPH_DEF_VERSION,
    min_consumer=versions.GRAPH_DEF_VERSION_MIN_CONSUMER,
    msg=None
)
```

assertRaises

```
assertRaises(
    excClass,
    callableObj=None,
    *args,
    **kwargs
)
```

Fail unless an exception of class excClass is raised by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception. If called with callableObj omitted or None, will return a context object used like this::

```
 with self.assertRaises(SomeException):
     do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object. The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion::

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesOpError

```
assertRaisesOpError(expected_err_re_or_predicate)
```

assertRaisesRegex

```
assertRaisesRegex(
    expected_exception,
    expected_regex,
    callable_obj=None,
    *args,
    **kwargs
)
```

Asserts that the message in a raised exception matches a regex.

Args:

- **`expected_exception`**: Exception class expected to be raised.
- **`expected_regex`**: Regex (re pattern object or string) expected to be found in error message.
- **`callable_obj`**: Function to be called.
- **`msg`**: Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.
- **`args`**: Extra args.
- **`kwargs`**: Extra kwargs.

assertRaisesRegexp

```
assertRaisesRegexp(
    expected_exception,
    expected_regex,
    callable_obj=None,
    *args,
    **kwargs
)
```

Asserts that the message in a raised exception matches a regex.

Args:

- **`expected_exception`**: Exception class expected to be raised.
- **`expected_regex`**: Regex (re pattern object or string) expected to be found in error message.
- **`callable_obj`**: Function to be called.
- **`msg`**: Optional message used in case of failure. Can only be used when assertRaisesRegex is used as a context manager.
- **`args`**: Extra args.
- **`kwargs`**: Extra kwargs.

assertRaisesWithLiteralMatch

```
assertRaisesWithLiteralMatch(
    expected_exception,
    expected_exception_message,
    callable_obj=None,
    *args,
    **kwargs
)
```

Asserts that the message in a raised exception equals the given string.
Unlike assertRaisesRegex, this method takes a literal string, not a regular expression.
with self.assertRaisesWithLiteralMatch(ExType, 'message'): DoSomething()

Args:

- **`expected_exception`**: Exception class expected to be raised.
- **`expected_exception_message`**: String message expected in the raised exception. For a raise exception e, expected_exception_message must equal str(e).
- **`callable_obj`**: Function to be called, or None to return a context.
- **`*args`**: Extra args.
- **`**kwargs`**: Extra kwargs.

Returns:
A context manager if callable_obj is None. Otherwise, None.

Raises:
self.failureException if callable_obj does not raise a matching exception.

assertRaisesWithPredicateMatch

```
assertRaisesWithPredicateMatch(
    *args,
    **kwds
)
```

Returns a context manager to enclose code expected to raise an exception.
If the exception is an OpError, the op stack is also included in the message predicate search.

Args:

- **`exception_type`**: The expected type of exception that should be raised.
- **`expected_err_re_or_predicate`**: If this is callable, it should be a function of one argument that inspects the passed-in exception and returns True (success) or False (please fail the test). Otherwise, the error message is expected to match this regular expression partially.

Returns:
A context manager to surround code that is expected to raise an exception.

assertRegex

```
assertRegex(
    text,
    expected_regex,
    msg=None
)
```

Fail the test unless the text matches the regular expression.

assertRegexMatch

```
assertRegexMatch(
    actual_str,
    regexes,
    message=None
)
```

Asserts that at least one regex in regexes matches str.
If possible you should use `assertRegex`, which is a simpler version of this method. `assertRegex`takes a single regular expression (a string or re compiled object) instead of a list.

Notes:

1. This function uses substring matching, i.e. the matching succeeds if *any* substring of the error message matches *any* regex in the list. This is more convenient for the user than full-string matching.
2. If regexes is the empty list, the matching will always fail.
3. Use regexes=[''] for a regex that will always pass.
4. '.' matches any single character *except* the newline. To match any character, use '(.|\n)'.
5. '^' matches the beginning of each line, not just the beginning of the string. Similarly, '$' matches the end of each line.
6. An exception will be thrown if regexes contains an invalid regex.

Args:

- **`actual_str`**: The string we try to match with the items in regexes.
- **`regexes`**: The regular expressions we want to match against str. See "Notes" above for detailed notes on how this is interpreted.
- **`message`**: The message to be printed if the test fails.

assertRegexpMatches

```
assertRegexpMatches(
    *args,
    **kwargs
)
```

assertSameElements

```
assertSameElements(
    expected_seq,
    actual_seq,
    msg=None
)
```

Asserts that two sequences have the same elements (in any order).
This method, unlike assertCountEqual, doesn't care about any duplicates in the expected and actual sequences.

```
assertSameElements([1, 1, 1, 0, 0, 0], [0, 1]) # Doesn't raise an AssertionError
```

If possible, you should use assertCountEqual instead of assertSameElements.

Args:

- **`expected_seq`**: A sequence containing elements we are expecting.
- **`actual_seq`**: The sequence that we are testing.
- **`msg`**: The message to be printed if the test fails.

assertSameStructure

```
assertSameStructure(
    a,
    b,
    aname='a',
    bname='b',
    msg=None
)
```

Asserts that two values contain the same structural content.
The two arguments should be data trees consisting of trees of dicts and lists. They will be deeply compared by walking into the contents of dicts and lists; other items will be compared using the == operator. If the two structures differ in content, the failure message will indicate the location within the structures where the first difference is found. This may be helpful when comparing large structures.
Mixed Sequence and Set types are supported. Mixed Mapping types are supported, but the order of the keys will not be considered in the comparison.

Args:

- **`a`**: The first structure to compare.
- **`b`**: The second structure to compare.

- `aname`: Variable name to use for the first structure in assertion messages.
- `bname`: Variable name to use for the second structure.
- `msg`: Additional text to include in the failure message.

assertSequenceAlmostEqual

```
assertSequenceAlmostEqual(
    expected_seq,
    actual_seq,
    places=None,
    msg=None,
    delta=None
)
```

An approximate equality assertion for ordered sequences.
Fail if the two sequences are unequal as determined by their value differences rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the difference between each value in the two sequences is more than the given delta.
Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).
If the two sequences compare equal then they will automatically compare almost equal.

Args:

- `expected_seq`: A sequence containing elements we are expecting.
- `actual_seq`: The sequence that we are testing.
- `places`: The number of decimal places to compare.
- `msg`: The message to be printed if the test fails.
- `delta`: The OK difference between compared values.

assertSequenceEqual

```
assertSequenceEqual(
    seq1,
    seq2,
    msg=None,
    seq_type=None
)
```

An equality assertion for ordered sequences (like lists and tuples).
For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Args:

- `seq1`: The first sequence to compare.
- `seq2`: The second sequence to compare.
- `seq_type`: The expected datatype of the sequences, or None if no datatype should be enforced.
- `msg`: Optional message to use on failure instead of a list of differences.

assertSequenceStartsWith

```
assertSequenceStartsWith(
    prefix,
    whole,
    msg=None
```

```
)
```

An equality assertion for the beginning of ordered sequences.
If prefix is an empty sequence, it will raise an error unless whole is also an empty sequence.
If prefix is not a sequence, it will raise an error if the first element of whole does not match.

Args:

- **`prefix`**: A sequence expected at the beginning of the whole parameter.
- **`whole`**: The sequence in which to look for prefix.
- **`msg`**: Optional message to report on failure.

assertSetEqual

```
assertSetEqual(
    set1,
    set2,
    msg=None
)
```

A set-specific equality assertion.

Args:

- **`set1`**: The first set to compare.
- **`set2`**: The second set to compare.
- **`msg`**: Optional message to use on failure instead of a list of differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertShapeEqual

```
assertShapeEqual(
    np_array,
    tf_tensor,
    msg=None
)
```

Asserts that a Numpy ndarray and a TensorFlow tensor have the same shape.

Args:

- **`np_array`**: A Numpy ndarray or Numpy scalar.
- **`tf_tensor`**: A Tensor.
- **`msg`**: Optional message to report on failure.

Raises:

- **`TypeError`**: If the arguments have the wrong type.

assertStartsWith

```
assertStartsWith(
    actual,
    expected_start,
    msg=None
)
```

Assert that actual.startswith(expected_start) is True.

Args:

- `actual`: str
- `expected_start`: str
- `msg`: Optional message to report on failure.

assertTotallyOrdered

```
assertTotallyOrdered(
    *groups,
    **kwargs
)
```

Asserts that total ordering has been implemented correctly.
For example, say you have a class A that compares only on its attribute x. Comparators other than **lt**are omitted for brevity.
class A(object): def **init**(self, x, y): self.x = x self.y = y
def **hash**(self): return hash(self.x)
def **lt**(self, other): try: return self.x < other.x except AttributeError: return NotImplemented
assertTotallyOrdered will check that instances can be ordered correctly. For example, self.assertTotallyOrdered([None], # None should come before everything else. [1], # Integers sort earlier. [A(1, 'a')], [A(2, 'b')], # 2 is after 1. [A(3, 'c'), A(3, 'd')], # The second argument is irrelevant. [A(4, 'z')], ['foo']) # Strings sort last.

Args:

- `*groups`: A list of groups of elements. Each group of elements is a list of objects that are equal. The elements in each group must be less than the elements in the group after it. For example, these groups are totally ordered: [None], [1], [2, 2], [3]. **kwargs: optional msg keyword argument can be passed.

assertTrue

```
assertTrue(
    expr,
    msg=None
)
```

Check that the expression is true.

assertTupleEqual

```
assertTupleEqual(
    tuple1,
    tuple2,
    msg=None
)
```

A tuple-specific equality assertion.

Args:

- `tuple1`: The first tuple to compare.
- `tuple2`: The second tuple to compare.
- `msg`: Optional message to use on failure instead of a list of differences.

assertUrlEqual

```
assertUrlEqual(
    a,
```

```
    b,
    msg=None
)
```

Asserts that urls are equal, ignoring ordering of query params.

assertWarns

```
assertWarns(
    expected_warning,
    callable_obj=None,
    *args,
    **kwargs
)
```

Fail unless a warning of class warnClass is triggered by callable_obj when invoked with arguments args and keyword arguments kwargs. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.
If called with callable_obj omitted or None, will return a context object used like this::

```
 with self.assertWarns(SomeWarning):
     do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object. The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion::

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex

```
assertWarnsRegex(
    expected_warning,
    expected_regex,
    callable_obj=None,
    *args,
    **kwargs
)
```

Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Args:

- **`expected_warning`**: Warning class expected to be triggered.
- **`expected_regex`**: Regex (re pattern object or string) expected to be found in error message.
- **`callable_obj`**: Function to be called.
- **`msg`**: Optional message used in case of failure. Can only be used when assertWarnsRegex is used as a context manager.

- **`args`**: Extra args.
- **`kwargs`**: Extra kwargs.

`assert_`

```
assert_(
    *args,
    **kwargs
)
```

`cached_session`

```
cached_session(
    *args,
    **kwds
)
```

Returns a TensorFlow Session for use in executing tests.
This method behaves differently than self.session(): for performance reasons `cached_session` will by default reuse the same session within the same test. The session returned by this function will only be closed at the end of the test (in the TearDown function).
Use the `use_gpu` and `force_gpu` options to control where ops are run. If `force_gpu` is True, all ops are pinned to `/device:GPU:0`. Otherwise, if `use_gpu` is True, TensorFlow tries to run as many ops on the GPU as possible. If both `force_gpu and`use_gpu` are False, all ops are pinned to the CPU.

Example:

```
class MyOperatorTest(test_util.TensorFlowTestCase):
  def testMyOperator(self):
    with self.cached_session(use_gpu=True) as sess:
      valid_input = [1.0, 2.0, 3.0, 4.0, 5.0]
      result = MyOperator(valid_input).eval()
      self.assertEqual(result, [1.0, 2.0, 3.0, 5.0, 8.0]
      invalid_input = [-1.0, 2.0, 7.0]
      with self.assertRaisesOpError("negative input not supported"):
        MyOperator(invalid_input).eval()
```

Args:

- **`graph`**: Optional graph to use during the returned session.
- **`config`**: An optional config_pb2.ConfigProto to use to configure the session.
- **`use_gpu`**: If True, attempt to run as many ops as possible on GPU.
- **`force_gpu`**: If True, pin all ops to `/device:GPU:0`.

Yields:

A Session object that should be used as a context manager to surround the graph building and execution code in a test case.

`captureWritesToStream`

```
captureWritesToStream(
    *args,
    **kwds
)
```

A context manager that captures the writes to a given stream.

This context manager captures all writes to a given stream inside of a `CapturedWrites` object. When this context manager is created, it yields the `CapturedWrites` object. The captured contents can be accessed by calling `.contents()` on the `CapturedWrites`.
For this function to work, the stream must have a file descriptor that can be modified using `os.dup`and `os.dup2`, and the stream must support a `.flush()` method. The default python sys.stdout and sys.stderr are examples of this. Note that this does not work in Colab or Jupyter notebooks, because those use alternate stdout streams.

Example:

```
class MyOperatorTest(test_util.TensorFlowTestCase):
  def testMyOperator(self):
    input = [1.0, 2.0, 3.0, 4.0, 5.0]
    with self.captureWritesToStream(sys.stdout) as captured:
      result = MyOperator(input).eval()
    self.assertStartsWith(captured.contents(), "This was printed.")
```

Args:

- **`stream`**: The stream whose writes should be captured. This stream must have a file descriptor, support writing via using that file descriptor, and must have a `.flush()` method.

Yields:

A `CapturedWrites` object that contains all writes to the specified stream made during this context.

checkedThread

```
checkedThread(
    target,
    args=None,
    kwargs=None
)
```

Returns a Thread wrapper that asserts 'target' completes successfully.
This method should be used to create all threads in test cases, as otherwise there is a risk that a thread will silently fail, and/or assertions made in the thread will not be respected.

Args:

- **`target`**: A callable object to be executed in the thread.
- **`args`**: The argument tuple for the target invocation. Defaults to ().
- **`kwargs`**: A dictionary of keyword arguments for the target invocation. Defaults to {}.

Returns:

A wrapper for threading.Thread that supports start() and join() methods.

countTestCases

```
countTestCases()
```

create_tempdir

```
create_tempdir(
    name=None,
    cleanup=None
)
```

Create a temporary directory specific to the test.

NOTE: The directory and its contents will be recursively cleared before creation. This ensures that there is no pre-existing state.
This creates a named directory on disk that is isolated to this test, and will be properly cleaned up by the test. This avoids several pitfalls of creating temporary directories for test purposes, as well as makes it easier to setup directories and verify their contents.
See also: `create_tempfile()` for creating temporary files.

Args:

- **`name`**: Optional name of the directory. If not given, a unique name will be generated and used.
- **`cleanup`**: Optional cleanup policy on when/if to remove the directory (and all its contents) at the end of the test. If None, then uses `self.tempfile_cleanup`.

Returns:
A _TempDir representing the created directory.

`create_tempfile`

```
create_tempfile(
    file_path=None,
    content=None,
    mode='w',
    encoding='utf8',
    errors='strict',
    cleanup=None
)
```

Create a temporary file specific to the test.
This creates a named file on disk that is isolated to this test, and will be properly cleaned up by the test. This avoids several pitfalls of creating temporary files for test purposes, as well as makes it easier to setup files, their data, read them back, and inspect them when a test fails.
NOTE: This will zero-out the file. This ensures there is no pre-existing state.
See also: `create_tempdir()` for creating temporary directories.

Args:

- **`file_path`**: Optional file path for the temp file. If not given, a unique file name will be generated and used. Slashes are allowed in the name; any missing intermediate directories will be created. NOTE: This path is the path that will be cleaned up, including any directories in the path, e.g., 'foo/bar/baz.txt' will `rm -r foo`.
- **`content`**: Optional string or bytes to initially write to the file. If not specified, then an empty file is created.
- **`mode`**: Mode string to use when writing content. Only used if `content` is non-empty.
- **`encoding`**: Encoding to use when writing string content. Only used if `content` is text.
- **`errors`**: How to handle text to bytes encoding errors. Only used if `content` is text.
- **`cleanup`**: Optional cleanup policy on when/if to remove the directory (and all its contents) at the end of the test. If None, then uses `self.tempfile_cleanup`.

Returns:
A _TempFile representing the created file.

`debug`

```
debug()
```

Run the test without collecting errors in a TestResult

defaultTestResult

```
defaultTestResult()
```

doCleanups

```
doCleanups()
```

Execute all cleanup functions. Normally called for you after tearDown.

evaluate

```
evaluate(tensors)
```

Evaluates tensors and returns numpy values.

Args:

- **`tensors`**: A Tensor or a nested list/tuple of Tensors.

Returns:
tensors numpy values.

fail

```
fail(
    msg=None,
    prefix=None
)
```

Fail immediately with the given message, optionally prefixed.

failIf

```
failIf(
    *args,
    **kwargs
)
```

failIfAlmostEqual

```
failIfAlmostEqual(
    *args,
    **kwargs
)
```

failIfEqual

```
failIfEqual(
    *args,
    **kwargs
)
```

failUnless

```
failUnless(
    *args,
```

```
    **kwargs
)
```

failUnlessAlmostEqual

```
failUnlessAlmostEqual(
    *args,
    **kwargs
)
```

failUnlessEqual

```
failUnlessEqual(
    *args,
    **kwargs
)
```

failUnlessRaises

```
failUnlessRaises(
    *args,
    **kwargs
)
```

get_temp_dir

```
get_temp_dir()
```

Returns a unique temporary directory for the test to use.
If you call this method multiple times during in a test, it will return the same folder. However, across different runs the directories will be different. This will ensure that across different runs tests will not be able to pollute each others environment. If you need multiple unique directories within a single test, you should use tempfile.mkdtemp as follows: tempfile.mkdtemp(dir=self.get_temp_dir()):

Returns:
string, the path to the unique temporary directory created for this test.

id

```
id()
```

run

```
run(result=None)
```

session

```
session(
    *args,
    **kwds
)
```

Returns a TensorFlow Session for use in executing tests.

Note that this will set this session and the graph as global defaults.
Use the `use_gpu` and `force_gpu` options to control where ops are run. If `force_gpu` is True, all ops are pinned to `/device:GPU:0`. Otherwise, if `use_gpu` is True, TensorFlow tries to run as many ops on the GPU as possible. If both `force_gpu and`use_gpu` are False, all ops are pinned to the CPU.

Example:

```
class MyOperatorTest(test_util.TensorFlowTestCase):
  def testMyOperator(self):
    with self.session(use_gpu=True):
      valid_input = [1.0, 2.0, 3.0, 4.0, 5.0]
      result = MyOperator(valid_input).eval()
      self.assertEqual(result, [1.0, 2.0, 3.0, 5.0, 8.0]
      invalid_input = [-1.0, 2.0, 7.0]
      with self.assertRaisesOpError("negative input not supported"):
        MyOperator(invalid_input).eval()
```

Args:

- **`graph`**: Optional graph to use during the returned session.
- **`config`**: An optional config_pb2.ConfigProto to use to configure the session.
- **`use_gpu`**: If True, attempt to run as many ops as possible on GPU.
- **`force_gpu`**: If True, pin all ops to `/device:GPU:0`.

Yields:

A Session object that should be used as a context manager to surround the graph building and execution code in a test case.

`setUp`

```
setUp()
```

`setUpClass`

```
setUpClass(cls)
```

Hook method for setting up class fixture before running tests in the class.

`shortDescription`

```
shortDescription()
```

Formats both the test method name and the first line of its docstring.
If no docstring is given, only returns the method name.
This method overrides unittest.TestCase.shortDescription(), which only returns the first line of the docstring, obscuring the name of the test upon failure.

Returns:

- **`desc`**: A short description of a test method.

`skipTest`

```
skipTest(reason)
```

Skip this test.

subTest

```
subTest(
    *args,
    **kwds
)
```

Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

tearDown

```
tearDown()
```

tearDownClass

```
tearDownClass(cls)
```

Hook method for deconstructing the class fixture after running all tests in the class.

test_session

```
test_session(
    graph=None,
    config=None,
    use_gpu=False,
    force_gpu=False
)
```

Use cached_session instead. (deprecated)
Warning: THIS FUNCTION IS DEPRECATED. It will be removed in a future version. Instructions for updating: Use `self.session()` or `self.cached_session()` instead.

Class Members

- `longMessage = True`
- `maxDiff = 1600`
- `tempfile_cleanup`

tf.test.TestCase.failureException

- Contents
- Class failureException
  - Aliases:
- __init__

Class `failureException`

Assertion failed.

Aliases:

- Class `tf.compat.v1.test.TestCase.failureException`
- Class `tf.compat.v2.test.TestCase.failureException`
- Class `tf.test.TestCase.failureException`

`__init__`

```
__init__(
    *args,
    **kwargs
)
```

Module: tf.compat.v1.tpu / tf.tpu

- **Contents**
- Modules
- Classes
- Functions

Ops related to Tensor Processing Units.

Modules

`experimental` module: Public API for tf.tpu.experimental namespace.

Classes

`class CrossShardOptimizer`: An optimizer that averages gradients across TPU shards.

Functions

`batch_parallel(...)`: Shards `computation` along the batch dimension for parallel execution.
`bfloat16_scope(...)`: Scope class for bfloat16 variables so that the model uses custom getter.
`core(...)`: Returns the device name for a core in a replicated TPU computation.
`cross_replica_sum(...)`: Sum the input tensor across replicas according to group_assignment.
`initialize_system(...)`: Initializes a distributed TPU system for use with TensorFlow.
`outside_compilation(...)`: Builds part of a computation outside any current TPU replicate scope.
`replicate(...)`: Builds a graph operator that runs a replicated TPU computation.
`rewrite(...)`: Rewrites `computation` for execution on a TPU system.
`shard(...)`: Shards `computation` for parallel execution.
`shutdown_system(...)`: Shuts down a running a distributed TPU system.

tf.compat.v1.tpu.batch_parallel

Shards `computation` along the batch dimension for parallel execution.

```
tf.compat.v1.tpu.batch_parallel(
    computation,
    inputs=None,
    num_shards=1,
    infeed_queue=None,
    device_assignment=None,
    name=None
)
```

Defined in `python/tpu/tpu.py`.
Convenience wrapper around shard().
`inputs` must be a list of Tensors or None (equivalent to an empty list). Each input is split into `num_shards` pieces along the 0-th dimension, and computation is applied to each shard in parallel.
Tensors are broadcast to all shards if they are lexically captured by `computation`. e.g.,
x = tf.constant(7) def computation(): return x + 3 ... = shard(computation, ...)
The outputs from all shards are concatenated back together along their 0-th dimension.
Inputs and outputs of the computation must be at least rank-1 Tensors.

Args:

- `computation`: A Python function that builds a computation to apply to each shard of the input.
- `inputs`: A list of input tensors or None (equivalent to an empty list). The 0-th dimension of each Tensor must have size divisible by `num_shards`.
- `num_shards`: The number of shards.
- `infeed_queue`: If not `None`, the `InfeedQueue` from which to append a tuple of arguments as inputs to `computation`.
- `device_assignment`: If not `None`, a `DeviceAssignment` describing the mapping between logical cores in the computation with physical cores in the TPU topology. Uses a default device assignment if `None`. The `DeviceAssignment` may be omitted if each shard of the computation uses only one core, and there is either only one shard, or the number of shards is equal to the number of cores in the TPU system.
- `name`: (Deprecated) Does nothing.

Returns:

A list of output tensors.

Raises:

- `ValueError`: If `num_shards <= 0`

tf.compat.v1.tpu.bfloat16_scope

Scope class for bfloat16 variables so that the model uses custom getter.

```
tf.compat.v1.tpu.bfloat16_scope()
```

Defined in `python/tpu/bfloat16.py`.

This enables variables to be read as bfloat16 type when using get_variable.

tf.compat.v1.tpu.core

Returns the device name for a core in a replicated TPU computation.

```
tf.compat.v1.tpu.core(num)
```

Defined in `python/tpu/tpu.py`.

Args:

- `num`: the virtual core number within each replica to which operators should be assigned.

Returns:

A device name, suitable for passing to `tf.device()`.

tf.compat.v1.tpu.CrossShardOptimizer

- Contents
- Class CrossShardOptimizer
- __init__
- Methods
  - apply_gradients

Class `CrossShardOptimizer`

An optimizer that averages gradients across TPU shards.

Inherits From: `Optimizer`

Defined in `python/tpu/tpu_optimizer.py`.

`__init__`

```
__init__(
    opt,
    reduction=losses.Reduction.MEAN,
    name='CrossShardOptimizer',
    group_assignment=None
)
```

Construct a new cross-shard optimizer.

Args:

- **`opt`**: An existing `Optimizer` to encapsulate.
- **`reduction`**: The reduction to apply to the shard losses.
- **`name`**: Optional name prefix for the operations created when applying gradients. Defaults to "CrossShardOptimizer".
- **`group_assignment`**: Optional 2d int32 lists with shape [num_groups, num_replicas_per_group] which describles how to apply optimizer to subgroups.

Raises:

- **`ValueError`**: If reduction is not a valid cross-shard reduction.

Methods

`apply_gradients`

```
apply_gradients(
    grads_and_vars,
    global_step=None,
    name=None
)
```

Apply gradients to variables.
Calls tpu_ops.cross_replica_sum() to sum gradient contributions across replicas, and then applies the real optimizer.

Args:

- **`grads_and_vars`**: List of (gradient, variable) pairs as returned by compute_gradients().
- **`global_step`**: Optional Variable to increment by one after the variables have been updated.
- **`name`**: Optional name for the returned operation. Default to the name passed to the Optimizer constructor.

Returns:

An `Operation` that applies the gradients. If `global_step` was not None, that operation also increments `global_step`.

Raises:

- **`ValueError`**: If the grads_and_vars is malformed.

`compute_gradients`

```
compute_gradients(
    loss,
    var_list=None,
    **kwargs
)
```

Compute gradients of "loss" for the variables in "var_list".

This simply wraps the compute_gradients() from the real optimizer. The gradients will be aggregated in the apply_gradients() so that user can modify the gradients like clipping with per replica global norm if needed. The global norm with aggregated gradients can be bad as one replica's huge gradients can hurt the gradients from other replicas.

Args:

- **`loss`**: A Tensor containing the value to minimize.
- **`var_list`**: Optional list or tuple of `tf.Variable` to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKey.TRAINABLE_VARIABLES`.
- **`**kwargs`**: Keyword arguments for compute_gradients().

Returns:

A list of (gradient, variable) pairs.

Raises:

- **`ValueError`**: If not within a tpu_shard_context or group_assignment is invalid.

get_name

```
get_name()
```

get_slot

```
get_slot(
    *args,
    **kwargs
)
```

Return a slot named "name" created for "var" by the Optimizer.
This simply wraps the get_slot() from the actual optimizer.

Args:

- **`*args`**: Arguments for get_slot().
- **`**kwargs`**: Keyword arguments for get_slot().

Returns:

The `Variable` for the slot if it was created, `None` otherwise.

get_slot_names

```
get_slot_names(
    *args,
    **kwargs
)
```

Return a list of the names of slots created by the `Optimizer`.
This simply wraps the get_slot_names() from the actual optimizer.

Args:

- **`*args`**: Arguments for get_slot().
- **`**kwargs`**: Keyword arguments for get_slot().

Returns:

A list of strings.

minimize

```
minimize(
    loss,
    global_step=None,
```

```
    var_list=None,
    gate_gradients=GATE_OP,
    aggregation_method=None,
    colocate_gradients_with_ops=False,
    name=None,
    grad_loss=None
)
```

Add operations to minimize `loss` by updating `var_list`.
This method simply combines calls `compute_gradients()` and `apply_gradients()`. If you want to process the gradient before applying them
call `compute_gradients()` and `apply_gradients()`explicitly instead of using this function.

Args:

- **`loss`**: A `Tensor` containing the value to minimize.
- **`global_step`**: Optional `Variable` to increment by one after the variables have been updated.
- **`var_list`**: Optional list or tuple of `Variable` objects to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.
- **`gate_gradients`**: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or`GATE_GRAPH`.
- **`aggregation_method`**: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- **`colocate_gradients_with_ops`**: If True, try colocating gradients with the corresponding op.
- **`name`**: Optional name for the returned operation.
- **`grad_loss`**: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

An Operation that updates the variables in `var_list`. If `global_step` was not `None`, that operation also increments `global_step`.

Raises:

- **`ValueError`**: If some of the variables are not `Variable` objects.

Eager Compatibility

When eager execution is enabled, `loss` should be a Python function that takes no arguments and computes the value to be minimized. Minimization (and gradient computation) is done with respect to the elements of `var_list` if not None, else with respect to any trainable variables created during the execution of
the `loss` function. `gate_gradients`, `aggregation_method`,`colocate_gradients_with_ops` and `grad_loss` are ignored when eager execution is enabled.

variables

```
variables()
```

Forwarding the variables from the underlying optimizer.

Class Members

- `GATE_GRAPH = 2`
- `GATE_NONE = 0`
- `GATE_OP = 1`

tf.compat.v1.tpu.cross_replica_sum

Sum the input tensor across replicas according to group_assignment.

```
tf.compat.v1.tpu.cross_replica_sum(
    x,
    group_assignment=None,
    name=None
)
```

Defined in `python/tpu/ops/tpu_ops.py`.

Args:

- **`x`**: The local tensor to the sum.
- **`group_assignment`**: Optional 2d int32 lists with shape [num_groups, num_replicas_per_group]. `group_assignment[i]` represents the replica ids in the ith subgroup.
- **`name`**: Optional op name.

Returns:

A `Tensor` which is summed across replicas.

tf.compat.v1.tpu.initialize_system

Initializes a distributed TPU system for use with TensorFlow.

```
tf.compat.v1.tpu.initialize_system(
    embedding_config=None,
    job=None
)
```

Defined in `python/tpu/tpu.py`.

Args:

- **`embedding_config`**: If not None, a `TPUEmbeddingConfiguration` proto describing the desired configuration of the hardware embedding lookup tables. If embedding_config is None, no hardware embeddings can be used.
- **`job`**: The job (the XXX in TensorFlow device specification /job:XXX) that contains the TPU devices that will be initialized. If job=None it is assumed there is only one job in the TensorFlow flock, and an error will be returned if this assumption does not hold.

Returns:

A serialized `TopologyProto` that describes the TPU system. Note: the topology must be evaluated using `Session.run` before it can be used.

tf.compat.v1.tpu.outside_compilation

Builds part of a computation outside any current TPU replicate scope.

```
tf.compat.v1.tpu.outside_compilation(
    computation,
    *args,
    **kwargs
)
```

Defined in `python/tpu/tpu.py`.

Args:

- **`computation`**: A Python function that builds the computation to place on the host.
- **`*args`**: the positional arguments for the computation.
- **`**kwargs`**: the keyword arguments for the computation.

Returns:
The Tensors returned by computation.

tf.compat.v1.tpu.replicate

Builds a graph operator that runs a replicated TPU computation.

```
tf.compat.v1.tpu.replicate(
    computation,
    inputs=None,
    infeed_queue=None,
    device_assignment=None,
    name=None,
    maximum_shapes=None
)
```

Defined in `python/tpu/tpu.py`.

Args:

- **`computation`**: A Python function that builds the computation to replicate.
- **`inputs`**: A list of lists of input tensors or `None` (equivalent to `[[]]`), indexed by `[replica_num][input_num]`. All replicas must have the same number of inputs. Each input can be a nested structure containing values that are convertible to tensors. Note that passing an N-dimension list of compatible values will result in a N-dimention list of scalar tensors rather than a single Rank-N tensors. If you need different behavior, convert part of inputs to tensors with `tf.convert_to_tensor`.
- **`infeed_queue`**: If not `None`, the `InfeedQueue` from which to append a tuple of arguments as inputs to computation.
- **`device_assignment`**: If not `None`, a `DeviceAssignment` describing the mapping between logical cores in the computation with physical cores in the TPU topology. Uses a default device assignment if `None`. The `DeviceAssignment` may be omitted if each replica of the computation uses only one core, and there is either only one replica, or the number of replicas is equal to the number of cores in the TPU system.
- **`name`**: (Deprecated) Does nothing.
- **`maximum_shapes`**: A nested structure of tf.TensorShape representing the shape to which the respective component of each input element in each replica should be padded. Any unknown dimensions (e.g. tf.compat.v1.Dimension(None) in a tf.TensorShape or -1 in a tensor-like object) will be padded to the maximum size of that dimension over all replicas. Note that if the input dimension is already static, we won't do padding on it and we require the maximum_shapes to have the same value or None on that dimension. The structure of `maximum_shapes` needs to be the same as `inputs[0]`.

Returns:
A list of outputs, indexed by `[replica_num]` each output can be a nested structure same as what computation() returns with a few exceptions.
Exceptions include: 1) None output: a NoOp would be returned which control-depends on computation. 2) Single value output: A tuple containing the value would be returned. 3) Operation-only outputs: a NoOp would be returned which control-depends on computation.
TODO(b/121383831): Investigate into removing these special cases.

Raises:

- **`ValueError`**: If all replicas do not have equal numbers of input tensors.
- **`ValueError`**: If the number of inputs per replica does not match the number of formal parameters to `computation`.
- **`ValueError`**: If the static `inputs` dimensions don't match with the values given in `maximum_shapes`.

- `ValueError`: If the structure of inputs per replica does not match the structure of `maximum_shapes`.

tf.compat.v1.tpu.rewrite

Rewrites `computation` for execution on a TPU system.

```
tf.compat.v1.tpu.rewrite(
    computation,
    inputs=None,
    infeed_queue=None,
    device_assignment=None,
    name=None
)
```

Defined in `python/tpu/tpu.py`.

Args:

- **`computation`**: A Python function that builds a computation to apply to the input. If the function takes n inputs, 'inputs' should be a list of n tensors.
  `computation` may return a list of operations and tensors. Tensors must come before operations in the returned list. The return value of `rewrite` is a list of tensors corresponding to the tensors from the output of `computation`.
  All `Operation`s constructed during `computation` will be executed when evaluating any of the returned output tensors, not just the ones returned.
- **`inputs`**: A list of input tensors or `None` (equivalent to an empty list). Each input can be a nested structure containing values that are convertible to tensors. Note that passing an N-dimension list of compatible values will result in a N-dimention list of scalar tensors rather than a single Rank-N tensors. If you need different behavior, convert part of inputs to tensors with `tf.convert_to_tensor`.
- **`infeed_queue`**: If not `None`, the `InfeedQueue` from which to append a tuple of arguments as inputs to `computation`.
- **`device_assignment`**: if not `None`, a `DeviceAssignment` describing the mapping between logical cores in the computation with physical cores in the TPU topology. May be omitted for a single-core computation, in which case the core attached to task 0, TPU device 0 is used.
- **`name`**: (Deprecated) Does nothing.

Returns:

Same data structure as if computation(*inputs) is called directly with some exceptions for correctness. Exceptions include: 1) None output: a NoOp would be returned which control-depends on computation. 2) Single value output: A tuple containing the value would be returned. 3) Operation-only outputs: a NoOp would be returned which control-depends on computation. TODO(b/121383831): Investigate into removing these special cases.

tf.compat.v1.tpu.shard

Shards `computation` for parallel execution.

```
tf.compat.v1.tpu.shard(
    computation,
    inputs=None,
    num_shards=1,
    input_shard_axes=None,
    outputs_from_all_shards=True,
    output_shard_axes=None,
    infeed_queue=None,
```

```
    device_assignment=None,
    name=None
)
```

Defined in `python/tpu/tpu.py`.

`inputs` must be a list of Tensors or None (equivalent to an empty list), each of which has a corresponding split axis (from `input_shard_axes`). Each input is split into `num_shards` pieces along the corresponding axis, and computation is applied to each shard in parallel.

Tensors are broadcast to all shards if they are lexically captured by `computation`. e.g.,

x = tf.constant(7) def computation(): return x + 3 ... = shard(computation, ...)

TODO(phawkins): consider adding support for broadcasting Tensors passed as inputs.

If `outputs_from_all_shards` is true, the outputs from all shards of `computation` are concatenated back together along their `output_shards_axes`. Otherwise, each output is taken from an arbitrary shard.

Inputs and outputs of the computation must be at least rank-1 Tensors.

Args:

- **`computation`**: A Python function that builds a computation to apply to each shard of the input.
- **`inputs`**: A list of input tensors or None (equivalent to an empty list). Each input tensor has a corresponding shard axes, given by `input_shard_axes`, which must have size divisible by `num_shards`.
- **`num_shards`**: The number of shards.
- **`input_shard_axes`**: A list of dimensions along which to shard `inputs`, or `None`. `None` means "shard all inputs along dimension 0". If not `None`, there must be one dimension per input.
- **`outputs_from_all_shards`**: Boolean or list of boolean. For each output, if `True`, outputs from all shards are concatenated along the corresponding `output_shard_axes` entry. Otherwise, each output is taken from an arbitrary shard. If the argument is a boolean, the argument's value is used for each output.
- **`output_shard_axes`**: A list of dimensions along which to concatenate the outputs of `computation`, or `None`. `None` means "concatenate all outputs along dimension 0". If not `None`, there must be one dimension per output. Ignored if `outputs_from_all_shards` is False.
- **`infeed_queue`**: If not `None`, the `InfeedQueue` to use to augment the inputs of `computation`.
- **`device_assignment`**: If not `None`, a `DeviceAssignment` describing the mapping between logical cores in the computation with physical cores in the TPU topology. Uses a default device assignment if `None`. The `DeviceAssignment` may be omitted if each shard of the computation uses only one core, and there is either only one shard, or the number of shards is equal to the number of cores in the TPU system.
- **`name`**: (Deprecated) Does nothing.

Returns:

A list of output tensors.

Raises:

- **`ValueError`**: If num_shards <= 0
- **`ValueError`**: If len(input_shard_axes) != len(inputs)
- **`ValueError`**: If len(output_shard_axes) != len(outputs from `computation`)

tf.compat.v1.tpu.shutdown_system

Shuts down a running a distributed TPU system.

```
tf.compat.v1.tpu.shutdown_system(job=None)
```

Defined in `python/tpu/tpu.py`.

Args:

- **`job`**: The job (the XXX in TensorFlow device specification /job:XXX) that contains the TPU devices that will be shutdown. If job=None it is assumed there is only one job in the TensorFlow flock, and an error will be returned if this assumption does not hold.

Module: tf.tpu.experimental

- **Contents**
- Classes
- Functions

Public API for tf.tpu.experimental namespace.

Classes

`class DeviceAssignment`: Mapping from logical cores in a computation to the physical TPU topology.

Functions

`initialize_tpu_system(...)`: Initialize the TPU devices.

tf.tpu.experimental.DeviceAssignment

- Contents
- Class DeviceAssignment
  - Aliases:
- __init__
- Properties

Class `DeviceAssignment`

Mapping from logical cores in a computation to the physical TPU topology.

Aliases:

- Class `tf.compat.v1.tpu.experimental.DeviceAssignment`
- Class `tf.compat.v2.tpu.experimental.DeviceAssignment`
- Class `tf.tpu.experimental.DeviceAssignment`

Defined in `python/tpu/device_assignment.py`.

Prefer to use the `DeviceAssignment.build()` helper to construct a `DeviceAssignment`; it is easier if less flexible than constructing a `DeviceAssignment` directly.

`__init__`

```
__init__(
    topology,
    core_assignment
)
```

Constructs a `DeviceAssignment` object.

Args:

- **`topology`**: A `Topology` object that describes the physical TPU topology.
- **`core_assignment`**: A logical to physical core mapping, represented as a rank 3 numpy array. See the description of the `core_assignment` property for more details.

Raises:

- **`ValueError`**: If `topology` is not `Topology` object.
- **`ValueError`**: If `core_assignment` is not a rank 3 numpy array.

Properties

`core_assignment`

The logical to physical core mapping.

Returns:
An integer numpy array of rank 3, with shape `[num_replicas, num_cores_per_replica, topology_rank]`. Maps (replica, logical core) pairs to physical topology coordinates.

`num_cores_per_replica`

The number of cores per replica.

`num_replicas`

The number of replicas of the computation.

`topology`

A `Topology` that describes the TPU topology.

Methods

`build`

```
@staticmethod
build(
    topology,
    computation_shape=None,
    computation_stride=None,
    num_replicas=1
)
```

`coordinates`

```
coordinates(
    replica,
    logical_core
)
```

Returns the physical topology coordinates of a logical core.

`host_device`

```
host_device(
    replica=0,
    logical_core=0,
    job=None
)
```

Returns the CPU device attached to a logical core.

`lookup_replicas`

```
lookup_replicas(
    task_id,
    logical_core
)
```

Lookup replica ids by task number and logical core.

Args:

- `task_id`: TensorFlow task number.
- `logical_core`: An integer, identifying a logical core.

Returns:

A sorted list of the replicas that are attached to that task and logical_core.

Raises:

- `ValueError`: If no replica exists in the task which contains the logical core.

tpu_device

```
tpu_device(
    replica=0,
    logical_core=0,
    job=None
)
```

Returns the name of the TPU device assigned to a logical core.

tpu_ordinal

```
tpu_ordinal(
    replica=0,
    logical_core=0
)
```

Returns the ordinal of the TPU device assigned to a logical core.

tf.tpu.experimental.initialize_tpu_system

- Contents
- Aliases:

Initialize the TPU devices.

Aliases:

- `tf.compat.v1.tpu.experimental.initialize_tpu_system`
- `tf.compat.v2.tpu.experimental.initialize_tpu_system`
- `tf.tpu.experimental.initialize_tpu_system`

```
tf.tpu.experimental.initialize_tpu_system(cluster_resolver=None)
```

Defined in `python/tpu/tpu_strategy_util.py`.

Args:

- `cluster_resolver`: A tf.distribute.cluster_resolver.TPUClusterResolver, which provides information about the TPU cluster.

Returns:

The tf.tpu.Topology object for the topology of the TPU cluster.

Raises:

- `RuntimeError`: If no TPU devices found for eager execution.

Module: tf.xla.experimental

- **Contents**
- Functions

Public API for tf.xla.experimental namespace.

Functions

`compile(...)`: Builds an operator that compiles and runs `computation` with XLA.
`jit_scope(...)`: Enable or disable JIT compilation of operators within the scope.

tf.xla.experimental.compile

- Contents
- Aliases:

Builds an operator that compiles and runs `computation` with XLA.

Aliases:

- `tf.compat.v1.xla.experimental.compile`
- `tf.compat.v2.xla.experimental.compile`
- `tf.xla.experimental.compile`

```
tf.xla.experimental.compile(
    computation,
    inputs=None
)
```

Defined in `python/compiler/xla/xla.py`.
NOTE: In eager mode, `computation` will have `@tf.function` semantics.

Args:

- **`computation`**: A Python function that builds a computation to apply to the input. If the function takes n inputs, 'inputs' should be a list of n tensors.
  `computation` may return a list of operations and tensors. Tensors must come before operations in the returned list. The return value of `compile` is a list of tensors corresponding to the tensors from the output of `computation`.
  All `Operation`s returned from `computation` will be executed when evaluating any of the returned output tensors.
- **`inputs`**: A list of inputs or `None` (equivalent to an empty list). Each input can be a nested structure containing values that are convertible to tensors. Note that passing an N-dimension list of compatible values will result in a N-dimension list of scalar tensors rather than a single Rank-N tensors. If you need different behavior, convert part of inputs to tensors with `tf.convert_to_tensor`.

Returns:

Same data structure as if computation(*inputs) is called directly with some exceptions for correctness. Exceptions include: 1) None output: a NoOp would be returned which control-depends on computation. 2) Single value output: A tuple containing the value would be returned. 3) Operation-only outputs: a NoOp would be returned which control-depends on computation. TODO(b/121383831): Investigate into removing these special cases.

Raises:

- **`RuntimeError`**: if called when eager execution is enabled.

tf.xla.experimental.jit_scope

- Contents
- Aliases:

Enable or disable JIT compilation of operators within the scope.

Aliases:

- `tf.compat.v1.xla.experimental.jit_scope`
- `tf.compat.v2.xla.experimental.jit_scope`

- `tf.xla.experimental.jit_scope`

```
tf.xla.experimental.jit_scope(
    *args,
    **kwds
)
```

NOTE: This is an experimental feature.
The compilation is a hint and only supported on a best-effort basis.

Example usage:
with tf.xla.experimental.jit_scope(): c = tf.matmul(a, b) # compiled with tf.xla.experimental.jit_scope(compile_ops=False): d = tf.matmul(a, c) # not compiled with tf.xla.experimental.jit_scope(compile_ops=lambda node_def: 'matmul' in node_def.op.lower()): e = tf.matmul(a, b) + d # matmul is compiled, the addition is not.
Example of separate_compiled_gradients: # In the example below, the computations for f, g and h will all be compiled # in separate scopes. with tf.xla.experimental.jit_scope(separate_compiled_gradients=True): f = tf.matmul(a, b) g = tf.gradients([f], [a, b], name='mygrads1') h = tf.gradients([f], [a, b], name='mygrads2')

Args:

- **`compile_ops`**: Whether to enable or disable compilation in the scope. Either a Python bool, or a callable that accepts the parameter `node_def` and returns a python bool.
- **`separate_compiled_gradients`**: If true put each gradient subgraph into a separate compilation scope. This gives fine-grained control over which portions of the graph will be compiled as a single unit. Compiling gradients separately may yield better performance for some graphs. The scope is named based on the scope of the forward computation as well as the name of the gradients. As a result, the gradients will be compiled in a scope that is separate from both the forward computation, and from other gradients.

Raises:

- **`RuntimeError`**: if called when eager execution is enabled.

Yields:
The current scope, enabling or disabling compilation.

tf.compat.v1.user_ops.my_fact

Example of overriding the generated code for an Op.

```
tf.compat.v1.user_ops.my_fact()
```

Defined in `python/user_ops/user_ops.py`.

Module: tf.math

- Contents
- About Segmentation
- Functions

Math Operations.
Note: Functions taking **`Tensor`** arguments can also take anything accepted by **`tf.convert_to_tensor`**.**Note:** Elementwise binary operations in TensorFlow follow numpy-style broadcasting.
TensorFlow provides a variety of math functions including:

- Basic arithmetic operators and trigonometric functions.
- Special math functions (like: `tf.math.igamma` and `tf.math.zeta`)
- Complex number functions (like: `tf.math.imag` and `tf.math.angle`)

- Reductions and scans (like: `tf.math.reduce_mean` and `tf.math.cumsum`)
- Segment functions (like: `tf.math.segment_sum`)

See: `tf.linalg` for matrix and tensor functions.

About Segmentation

TensorFlow provides several operations that you can use to perform common math computations on tensor segments. Here a segmentation is a partitioning of a tensor along the first dimension, i.e. it defines a mapping from the first dimension onto `segment_ids`. The `segment_ids` tensor should be the size of the first dimension, `d0`, with consecutive IDs in the range `0` to `k`, where `k<d0`. In particular, a segmentation of a matrix tensor is a mapping of rows to segments.

For example:

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])
tf.math.segment_sum(c, tf.constant([0, 0, 1]))
#  ==>  [[0 0 0 0]
#        [5 6 7 8]]
```

The standard `segment_*` functions assert that the segment indices are sorted. If you have unsorted indices use the equivalent `unsorted_segment_` function. Thses functions take an additional argument `num_segments` so that the output tensor can be efficiently allocated.

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])
tf.math.unsorted_segment_sum(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 6,  8, 10, 12],
#       [-1, -2, -3, -4]]
```

Functions

`abs(...)`: Computes the absolute value of a tensor.
`accumulate_n(...)`: Returns the element-wise sum of a list of tensors.
`acos(...)`: Computes acos of x element-wise.
`acosh(...)`: Computes inverse hyperbolic cosine of x element-wise.
`add(...)`: Returns x + y element-wise.
`add_n(...)`: Adds all input tensors element-wise.
`angle(...)`: Returns the element-wise argument of a complex (or real) tensor.
`argmax(...)`: Returns the index with the largest value across axes of a tensor.
`argmin(...)`: Returns the index with the smallest value across axes of a tensor.
`asin(...)`: Computes the trignometric inverse sine of x element-wise.
`asinh(...)`: Computes inverse hyperbolic sine of x element-wise.
`atan(...)`: Computes the trignometric inverse tangent of x element-wise.
`atan2(...)`: Computes arctangent of `y/x` element-wise, respecting signs of the arguments.
`atanh(...)`: Computes inverse hyperbolic tangent of x element-wise.
`bessel_i0(...)`: Computes the Bessel i0 function of `x` element-wise.
`bessel_i0e(...)`: Computes the Bessel i0e function of `x` element-wise.
`bessel_i1(...)`: Computes the Bessel i1 function of `x` element-wise.
`bessel_i1e(...)`: Computes the Bessel i1e function of `x` element-wise.
`betainc(...)`: Compute the regularized incomplete beta integral Ix(a,b).
`bincount(...)`: Counts the number of occurrences of each value in an integer array.
`ceil(...)`: Returns element-wise smallest integer not less than x.
`confusion_matrix(...)`: Computes the confusion matrix from predictions and labels.
`conj(...)`: Returns the complex conjugate of a complex number.
`cos(...)`: Computes cos of x element-wise.
`cosh(...)`: Computes hyperbolic cosine of x element-wise.

`count_nonzero(...)`: Computes number of nonzero elements across dimensions of a tensor.
`cumprod(...)`: Compute the cumulative product of the tensor `x` along `axis`.
`cumsum(...)`: Compute the cumulative sum of the tensor `x` along `axis`.
`digamma(...)`: Computes Psi, the derivative of Lgamma (the log of the absolute value of
`divide(...)`: Computes Python style division of `x` by `y`.
`divide_no_nan(...)`: Computes an unsafe divide which returns 0 if the y is zero.
`equal(...)`: Returns the truth value of (x == y) element-wise.
`erf(...)`: Computes the Gauss error function of `x` element-wise.
`erfc(...)`: Computes the complementary error function of `x` element-wise.
`exp(...)`: Computes exponential of x element-wise. y=ex.
`expm1(...)`: Computes exponential of x - 1 element-wise.
`floor(...)`: Returns element-wise largest integer not greater than x.
`floordiv(...)`: Divides `x / y` elementwise, rounding toward the most negative integer.
`floormod(...)`: Returns element-wise remainder of division. When `x < 0` xor `y < 0` is
`greater(...)`: Returns the truth value of (x > y) element-wise.
`greater_equal(...)`: Returns the truth value of (x >= y) element-wise.
`igamma(...)`: Compute the lower regularized incomplete Gamma function `P(a, x)`.
`igammac(...)`: Compute the upper regularized incomplete Gamma function `Q(a, x)`.
`imag(...)`: Returns the imaginary part of a complex (or real) tensor.
`in_top_k(...)`: Says whether the targets are in the top `K` predictions.
`invert_permutation(...)`: Computes the inverse permutation of a tensor.
`is_finite(...)`: Returns which elements of x are finite.
`is_inf(...)`: Returns which elements of x are Inf.
`is_nan(...)`: Returns which elements of x are NaN.
`is_non_decreasing(...)`: Returns `True` if `x` is non-decreasing.
`is_strictly_increasing(...)`: Returns `True` if `x` is strictly increasing.
`l2_normalize(...)`: Normalizes along dimension `axis` using an L2 norm.
`lbeta(...)`: Computes ln(|Beta(x)|), reducing along the last dimension.
`less(...)`: Returns the truth value of (x < y) element-wise.
`less_equal(...)`: Returns the truth value of (x <= y) element-wise.
`lgamma(...)`: Computes the log of the absolute value of `Gamma(x)` element-wise.
`log(...)`: Computes natural logarithm of x element-wise.
`log1p(...)`: Computes natural logarithm of (1 + x) element-wise.
`log_sigmoid(...)`: Computes log sigmoid of `x` element-wise.
`log_softmax(...)`: Computes log softmax activations.
`logical_and(...)`: Returns the truth value of x AND y element-wise.
`logical_not(...)`: Returns the truth value of NOT x element-wise.
`logical_or(...)`: Returns the truth value of x OR y element-wise.
`logical_xor(...)`: Logical XOR function.
`maximum(...)`: Returns the max of x and y (i.e. x > y ? x : y) element-wise.
`minimum(...)`: Returns the min of x and y (i.e. x < y ? x : y) element-wise.
`mod(...)`: Returns element-wise remainder of division. When `x < 0` xor `y < 0` is
`multiply(...)`: Returns x * y element-wise.
`multiply_no_nan(...)`: Computes the product of x and y and returns 0 if the y is zero, even if x is NaN or infinite.
`negative(...)`: Computes numerical negative value element-wise.
`nextafter(...)`: Returns the next representable value of `x1` in the direction of `x2`, element-wise.
`not_equal(...)`: Returns the truth value of (x != y) element-wise.
`polygamma(...)`: Compute the polygamma function ψ(n)(x).
`polyval(...)`: Computes the elementwise value of a polynomial.
`pow(...)`: Computes the power of one value to another.

`real(...)`: Returns the real part of a complex (or real) tensor.
`reciprocal(...)`: Computes the reciprocal of x element-wise.
`reduce_all(...)`: Computes the "logical and" of elements across dimensions of a tensor.
`reduce_any(...)`: Computes the "logical or" of elements across dimensions of a tensor.
`reduce_euclidean_norm(...)`: Computes the Euclidean norm of elements across dimensions of a tensor.
`reduce_logsumexp(...)`: Computes log(sum(exp(elements across dimensions of a tensor))).
`reduce_max(...)`: Computes the maximum of elements across dimensions of a tensor.
`reduce_mean(...)`: Computes the mean of elements across dimensions of a tensor.
`reduce_min(...)`: Computes the minimum of elements across dimensions of a tensor.
`reduce_prod(...)`: Computes the product of elements across dimensions of a tensor.
`reduce_std(...)`: Computes the standard deviation of elements across dimensions of a tensor.
`reduce_sum(...)`: Computes the sum of elements across dimensions of a tensor.
`reduce_variance(...)`: Computes the variance of elements across dimensions of a tensor.
`rint(...)`: Returns element-wise integer closest to x.
`round(...)`: Rounds the values of a tensor to the nearest integer, element-wise.
`rsqrt(...)`: Computes reciprocal of square root of x element-wise.
`scalar_mul(...)`: Multiplies a scalar times a `Tensor` or `IndexedSlices` object.
`segment_max(...)`: Computes the maximum along segments of a tensor.
`segment_mean(...)`: Computes the mean along segments of a tensor.
`segment_min(...)`: Computes the minimum along segments of a tensor.
`segment_prod(...)`: Computes the product along segments of a tensor.
`segment_sum(...)`: Computes the sum along segments of a tensor.
`sigmoid(...)`: Computes sigmoid of `x` element-wise.
`sign(...)`: Returns an element-wise indication of the sign of a number.
`sin(...)`: Computes sin of x element-wise.
`sinh(...)`: Computes hyperbolic sine of x element-wise.
`softmax(...)`: Computes softmax activations.
`softplus(...)`: Computes softplus: `log(exp(features) + 1)`.
`softsign(...)`: Computes softsign: `features / (abs(features) + 1)`.
`sqrt(...)`: Computes square root of x element-wise.
`square(...)`: Computes square of x element-wise.
`squared_difference(...)`: Returns (x - y)(x - y) element-wise.
`subtract(...)`: Returns x - y element-wise.
`tan(...)`: Computes tan of x element-wise.
`tanh(...)`: Computes hyperbolic tangent of `x` element-wise.
`top_k(...)`: Finds values and indices of the `k` largest entries for the last dimension.
`truediv(...)`: Divides x / y elementwise (using Python 3 division operator semantics).
`unsorted_segment_max(...)`: Computes the maximum along segments of a tensor.
`unsorted_segment_mean(...)`: Computes the mean along segments of a tensor.
`unsorted_segment_min(...)`: Computes the minimum along segments of a tensor.
`unsorted_segment_prod(...)`: Computes the product along segments of a tensor.
`unsorted_segment_sqrt_n(...)`: Computes the sum along segments of a tensor divided by the sqrt(N).
`unsorted_segment_sum(...)`: Computes the sum along segments of a tensor.
`xdivy(...)`: Returns 0 if x == 0, and x / y otherwise, elementwise.
`xlogy(...)`: Returns 0 if x == 0, and x * log(y) otherwise, elementwise.
`zero_fraction(...)`: Returns the fraction of zeros in `value`.
`zeta(...)`: Compute the Hurwitz zeta function ζ(x,q).

tf.compat.v1.math.log_softmax

- Contents
- Aliases:

Computes log softmax activations. (deprecated arguments)

Aliases:

- `tf.compat.v1.math.log_softmax`
- `tf.compat.v1.nn.log_softmax`

```
tf.compat.v1.math.log_softmax(
    logits,
    axis=None,
    name=None,
    dim=None
)
```

Defined in `python/ops/nn_ops.py`.

Warning: SOME ARGUMENTS ARE DEPRECATED: `(dim)`. They will be removed in a future version. Instructions for updating: dim is deprecated, use axis instead

For each batch `i` and class `j` we have

```
logsoftmax = logits - log(reduce_sum(exp(logits), axis))
```

Args:

- **`logits`**: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- **`axis`**: The dimension softmax would be performed on. The default is -1 which indicates the last dimension.
- **`name`**: A name for the operation (optional).
- **`dim`**: Deprecated alias for `axis`.

Returns:

A `Tensor`. Has the same type as `logits`. Same shape as `logits`.

Raises:

- **`InvalidArgumentError`**: if `logits` is empty or `axis` is beyond the last dimension of `logits`.

tf.compat.v1.math.softmax

- Contents
- Aliases:

Computes softmax activations. (deprecated arguments)

Aliases:

- `tf.compat.v1.math.softmax`
- `tf.compat.v1.nn.softmax`

```
tf.compat.v1.math.softmax(
    logits,
    axis=None,
    name=None,
    dim=None
)
```

Defined in `python/ops/nn_ops.py`.

Warning: SOME ARGUMENTS ARE DEPRECATED: `(dim)`. They will be removed in a future version. Instructions for updating: dim is deprecated, use axis instead

This function performs the equivalent of

```
softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis)
```

Args:

- `logits`: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `axis`: The dimension softmax would be performed on. The default is -1 which indicates the last dimension.
- `name`: A name for the operation (optional).
- `dim`: Deprecated alias for `axis`.

Returns:

A `Tensor`. Has the same type and shape as `logits`.

Raises:

- `InvalidArgumentError`: if `logits` is empty or `axis` is beyond the last dimension of `logits`.

tf.math.abs

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes the absolute value of a tensor.

Aliases:

- `tf.RaggedTensor.__abs__`
- `tf.Tensor.__abs__`
- `tf.abs`
- `tf.compat.v1.RaggedTensor.__abs__`
- `tf.compat.v1.Tensor.__abs__`
- `tf.compat.v1.abs`
- `tf.compat.v1.math.abs`
- `tf.compat.v2.RaggedTensor.__abs__`
- `tf.compat.v2.Tensor.__abs__`
- `tf.compat.v2.abs`
- `tf.compat.v2.math.abs`
- `tf.math.abs`

```
tf.math.abs(
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Training checkpoints

Used in the tutorials:

- Pix2Pix
- tf.function

Given a tensor of integer or floating-point values, this operation returns a tensor of the same type, where each element contains the absolute value of the corresponding element in the input.

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 a2+b2. 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 `float16`, `float32`, `float64`, `int32`, `int64`, `complex64` or `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.abs(x.values, ...), x.dense_shape)`

tf.math.accumulate_n

- Contents
- Aliases:

Returns the element-wise sum of a list of tensors.

Aliases:

- `tf.compat.v1.accumulate_n`
- `tf.compat.v1.math.accumulate_n`
- `tf.compat.v2.math.accumulate_n`
- `tf.math.accumulate_n`

```
tf.math.accumulate_n(
    inputs,
    shape=None,
    tensor_dtype=None,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Optionally, pass `shape` and `tensor_dtype` for shape and type checking, otherwise, these are inferred.

`accumulate_n` performs the same operation as `tf.math.add_n`, but does not wait for all of its inputs to be ready before beginning to sum. This approach can save memory if inputs are ready at different times, since minimum temporary storage is proportional to the output size rather than the inputs' size.

`accumulate_n` is differentiable (but wasn't previous to TensorFlow 1.7).

For example:

```
a = tf.constant([[1, 2], [3, 4]])
b = tf.constant([[5, 0], [0, 6]])
tf.math.accumulate_n([a, b, a])  # [[7, 4], [6, 14]]

# Explicitly pass shape and type
tf.math.accumulate_n([a, b, a], shape=[2, 2], tensor_dtype=tf.int32)
```

```
                                                            # [[7,  4],
                                                            #  [6, 14]]
```

Args:

- **`inputs`**: A list of `Tensor` objects, each with same shape and type.
- **`shape`**: Expected shape of elements of `inputs` (optional). Also controls the output shape of this op, which may affect type inference in other ops. A value of `None` means "infer the input shape from the shapes in `inputs`".
- **`tensor_dtype`**: Expected data type of `inputs` (optional). A value of `None` means "infer the input dtype from `inputs[0]`".
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of same shape and type as the elements of `inputs`.

Raises:

- **`ValueError`**: If `inputs` don't all have same shape and dtype or the shape cannot be inferred.

tf.math.acos

- Contents
- Aliases:

Computes acos of x element-wise.

Aliases:

- `tf.acos`
- `tf.compat.v1.acos`
- `tf.compat.v1.math.acos`
- `tf.compat.v2.acos`
- `tf.compat.v2.math.acos`
- `tf.math.acos`

```
tf.math.acos(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.acosh

- Contents
- Aliases:

Computes inverse hyperbolic cosine of x element-wise.

Aliases:

- `tf.acosh`
- `tf.compat.v1.acosh`

- `tf.compat.v1.math.acosh`
- `tf.compat.v2.acosh`
- `tf.compat.v2.math.acosh`
- `tf.math.acosh`

```
tf.math.acosh(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.add

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Returns x + y element-wise.

Aliases:

- `tf.RaggedTensor.__add__`
- `tf.add`
- `tf.compat.v1.RaggedTensor.__add__`
- `tf.compat.v1.add`
- `tf.compat.v1.math.add`
- `tf.compat.v2.RaggedTensor.__add__`
- `tf.compat.v2.add`
- `tf.compat.v2.math.add`
- `tf.math.add`

```
tf.math.add(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the guide:

- Eager essentials
- Ragged Tensors

Used in the tutorials:

- Tensors and Operations

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

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `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`.

tf.math.add_n

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Adds all input tensors element-wise.

Aliases:

- `tf.add_n`
- `tf.compat.v1.add_n`
- `tf.compat.v1.math.add_n`
- `tf.compat.v2.add_n`
- `tf.compat.v2.math.add_n`
- `tf.math.add_n`

```
tf.math.add_n(
    inputs,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Convert Your Existing Code to TensorFlow 2.0
- Using GPUs

Used in the tutorials:

- Neural style transfer

Converts `IndexedSlices` objects into dense tensors prior to adding.

`tf.math.add_n` performs the same operation as `tf.math.accumulate_n`, but it waits for all of its inputs to be ready before beginning to sum. This buffering can result in higher memory consumption when inputs are ready at different times, since the minimum temporary storage required is proportional to the input size rather than the output size.

This op does not broadcast its inputs. If you need broadcasting, use `tf.math.add` (or the `+` operator) instead.

For example:

```
a = tf.constant([[3, 5], [4, 8]])
b = tf.constant([[1, 6], [2, 9]])
tf.math.add_n([a, b, a])  # [[7, 16], [10, 25]]
```

Args:

- **`inputs`**: A list of `tf.Tensor` or `tf.IndexedSlices` objects, each with same shape and type.

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

Returns:
A `Tensor` of same shape and type as the elements of `inputs`.

Raises:
- **ValueError**: If `inputs` don't all have same shape and dtype or the shape cannot be inferred.

tf.math.angle

- Contents
- Aliases:

Returns the element-wise argument of a complex (or real) tensor.

Aliases:
- `tf.compat.v1.angle`
- `tf.compat.v1.math.angle`
- `tf.compat.v2.math.angle`
- `tf.math.angle`

```
tf.math.angle(
    input,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Given a tensor `input`, this operation returns a tensor of type `float` that is the argument of each element in `input` considered as a complex number.
The elements in `input` are considered to be complex numbers of the form a+bj, where *a* is the real part and *b* is the imaginary part. If `input` is real then *b* is zero by definition.
The argument returned by this function is of the form atan2(b,a). If `input` is real, a tensor of all zeros is returned.

For example:

```
input = tf.constant([-2.25 + 4.75j, 3.25 + 5.75j], dtype=tf.complex64)
tf.math.angle(input).numpy()
# ==> array([2.0131705, 1.056345 ], dtype=float32)
```

Args:
- **input**: A `Tensor`. Must be one of the following types: `float`, `double`, `complex64`, `complex128`.
- **name**: A name for the operation (optional).

Returns:
A `Tensor` of type `float32` or `float64`.

tf.math.argmax

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Returns the index with the largest value across axes of a tensor.

Aliases:
- `tf.argmax`
- `tf.compat.v2.argmax`

- `tf.compat.v2.math.argmax`
- `tf.math.argmax`

```
tf.math.argmax(
    input,
    axis=None,
    output_type=tf.dtypes.int64,
    name=None
)

```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Convert Your Existing Code to TensorFlow 2.0
- Training and Evaluation with TensorFlow Keras

Used in the tutorials:

- Custom training: walkthrough
- Image Captioning with Attention
- Neural Machine Translation with Attention
- Transformer model for language understanding

Note that in case of ties the identity of the return value is not guaranteed.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`axis`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. int32 or int64, must be in the range `-rank(input), rank(input))`. Describes which axis of the input Tensor to reduce across. For vectors, use axis = 0.
- **`output_type`**: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `output_type`.

Usage:

```
import tensorflow as tf
a = [1, 10, 26.9, 2.8, 166.32, 62.3]
b = tf.math.argmax(input = a)
c = tf.keras.backend.eval(b)
# c = 4
# here a[4] = 166.32 which is the largest element of a across axis 0
```

tf.math.argmin

- Contents
- Aliases:

Returns the index with the smallest value across axes of a tensor.

Aliases:

- `tf.argmin`
- `tf.compat.v2.argmin`
- `tf.compat.v2.math.argmin`
- `tf.math.argmin`

```
tf.math.argmin(
    input,
    axis=None,
    output_type=tf.dtypes.int64,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Note that in case of ties the identity of the return value is not guaranteed.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `b` `float16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`axis`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. int32 or int64, must be in the range `-rank(input), rank(input))`. Describes which axis of the input Tensor to reduce across. For vectors, use axis = 0.
- **`output_type`**: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `output_type`.

Usage:

```
import tensorflow as tf
a = [1, 10, 26.9, 2.8, 166.32, 62.3]
b = tf.math.argmin(input = a)
c = tf.keras.backend.eval(b)
# c = 0
# here a[0] = 1 which is the smallest element of a across axis 0
```

tf.math.asin

- Contents
- Aliases:

Computes the trignometric inverse sine of x element-wise.

Aliases:

- `tf.asin`
- `tf.compat.v1.asin`
- `tf.compat.v1.math.asin`
- `tf.compat.v2.asin`
- `tf.compat.v2.math.asin`
- `tf.math.asin`

```
tf.math.asin(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
The `tf.math.asin` operation returns the inverse of `tf.math.sin`, such that if `y = tf.math.sin(x)` then, `x = tf.math.asin(y)`.
Note: The output of `tf.math.asin` will lie within the invertible range of sine, i.e [-pi/2, pi/2].

For example:

```
# Note: [1.047, 0.785] ~= [(pi/3), (pi/4)]
x = tf.constant([1.047, 0.785])
y = tf.math.sin(x) # [0.8659266, 0.7068252]

tf.math.asin(y) # [1.047, 0.785] = x
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.asinh

- Contents
- Aliases:

Computes inverse hyperbolic sine of x element-wise.

Aliases:

- `tf.asinh`
- `tf.compat.v1.asinh`
- `tf.compat.v1.math.asinh`
- `tf.compat.v2.asinh`
- `tf.compat.v2.math.asinh`
- `tf.math.asinh`

```
tf.math.asinh(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.atan

- Contents
- Aliases:

Computes the trignometric inverse tangent of x element-wise.

Aliases:

- `tf.atan`
- `tf.compat.v1.atan`
- `tf.compat.v1.math.atan`
- `tf.compat.v2.atan`

- `tf.compat.v2.math.atan`
- `tf.math.atan`

```
tf.math.atan(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
The `tf.math.atan` operation returns the inverse of `tf.math.tan`, such that if `y = tf.math.tan(x)` then, `x = tf.math.atan(y)`.
Note: The output of `tf.math.atan` will lie within the invertible range of tan, i.e (-pi/2, pi/2).

For example:

```
# Note: [1.047, 0.785] ~= [(pi/3), (pi/4)]
x = tf.constant([1.047, 0.785])
y = tf.math.tan(x) # [1.731261, 0.99920404]

tf.math.atan(y) # [1.047, 0.785] = x
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.atan2

- Contents
- Aliases:

Computes arctangent of `y/x` element-wise, respecting signs of the arguments.

Aliases:

- `tf.atan2`
- `tf.compat.v1.atan2`
- `tf.compat.v1.math.atan2`
- `tf.compat.v2.atan2`
- `tf.compat.v2.math.atan2`
- `tf.math.atan2`

```
tf.math.atan2(
    y,
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
This is the angle (\theta \in [-\pi, \pi]) such that [x = r \cos(\theta)] and [y = r \sin(\theta)] where (r = \sqrt(x^2 + y^2)).

Args:

- **`y`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`x`**: A `Tensor`. Must have the same type as `y`.

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

Returns:
A `Tensor`. Has the same type as `y`.

tf.math.atanh

- Contents
- Aliases:

Computes inverse hyperbolic tangent of x element-wise.

Aliases:

- `tf.atanh`
- `tf.compat.v1.atanh`
- `tf.compat.v1.math.atanh`
- `tf.compat.v2.atanh`
- `tf.compat.v2.math.atanh`
- `tf.math.atanh`

```
tf.math.atanh(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- `x`: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name`: A name for the operation (optional).

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

tf.math.bessel_i0

- Contents
- Aliases:

Computes the Bessel i0 function of `x` element-wise.

Aliases:

- `tf.compat.v1.math.bessel_i0`
- `tf.compat.v2.math.bessel_i0`
- `tf.math.bessel_i0`

```
tf.math.bessel_i0(
    x,
    name=None
)
```

Defined in `python/ops/special_math_ops.py`.
Modified Bessel function of order 0.
It is preferable to use the numerically stabler function `i0e(x)` instead.

Args:

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

Returns:
A `Tensor` or `SparseTensor`, respectively. Has the same type as `x`.

Scipy Compatibility
Equivalent to scipy.special.i0

tf.math.bessel_i0e

- Contents
- Aliases:

Computes the Bessel i0e function of `x` element-wise.

Aliases:

- `tf.compat.v1.math.bessel_i0e`
- `tf.compat.v2.math.bessel_i0e`
- `tf.math.bessel_i0e`

```
tf.math.bessel_i0e(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Exponentially scaled modified Bessel function of order 0 defined as `bessel_i0e(x) = exp(-abs(x)) bessel_i0(x)`.
This function is faster and numerically stabler than `bessel_i0(x)`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has the same type as `x`.
If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.bessel_i0e(x.values, ...), x.dense_shape)`

tf.math.bessel_i1

- Contents
- Aliases:

Computes the Bessel i1 function of `x` element-wise.

Aliases:

- `tf.compat.v1.math.bessel_i1`
- `tf.compat.v2.math.bessel_i1`
- `tf.math.bessel_i1`

```
tf.math.bessel_i1(
    x,
    name=None
)
```

Defined in `python/ops/special_math_ops.py`.
Modified Bessel function of order 1.
It is preferable to use the numerically stabler function `i1e(x)` instead.

Args:

- **`x`**: A `Tensor` or `SparseTensor`. Must be one of the following types: `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` or `SparseTensor`, respectively. Has the same type as `x`.

Scipy Compatibility

Equivalent to scipy.special.i1

tf.math.bessel_i1e

- Contents
- Aliases:

Computes the Bessel i1e function of `x` element-wise.

Aliases:

- `tf.compat.v1.math.bessel_i1e`
- `tf.compat.v2.math.bessel_i1e`
- `tf.math.bessel_i1e`

```
tf.math.bessel_i1e(
    x,
    name=None
)

```

Defined in generated file: `python/ops/gen_math_ops.py`.

Exponentially scaled modified Bessel function of order 0 defined as `bessel_i1e(x) = exp(-abs(x)) bessel_i1(x)`.

This function is faster and numerically stabler than `bessel_i1(x)`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.bessel_i1e(x.values, ...), x.dense_shape)`

tf.math.betainc

- Contents
- Aliases:

Compute the regularized incomplete beta integral Ix(a,b).

Aliases:

- `tf.compat.v1.betainc`
- `tf.compat.v1.math.betainc`
- `tf.compat.v2.math.betainc`
- `tf.math.betainc`

```
tf.math.betainc(
    a,
    b,
    x,
    name=None
```

```
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
The regularized incomplete beta integral is defined as:
Ix(a,b)=B(x;a,b)B(a,b)
where
B(x;a,b)=∫0xta−1(1−t)b−1dt
is the incomplete beta function and B(a,b) is the *complete* beta function.

Args:

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

Returns:
A `Tensor`. Has the same type as `a`.

tf.math.bincount

- Contents
- Aliases:

Counts the number of occurrences of each value in an integer array.

Aliases:

- `tf.compat.v2.math.bincount`
- `tf.math.bincount`

```
tf.math.bincount(
    arr,
    weights=None,
    minlength=None,
    maxlength=None,
    dtype=tf.dtypes.int32,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
If `minlength` and `maxlength` are not given, returns a vector with length `tf.reduce_max(arr) + 1`if `arr` is non-empty, and length 0 otherwise. If `weights` are non-None, then index `i` of the output stores the sum of the value in `weights` at each index where the corresponding value in `arr` is `i`.

Args:

- **`arr`**: An int32 tensor of non-negative values.
- **`weights`**: If non-None, must be the same shape as arr. For each value in `arr`, the bin will be incremented by the corresponding weight instead of 1.
- **`minlength`**: If given, ensures the output has length at least `minlength`, padding with zeros at the end if necessary.
- **`maxlength`**: If given, skips values in `arr` that are equal or greater than `maxlength`, ensuring that the output has length at most `maxlength`.
- **`dtype`**: If `weights` is None, determines the type of the output bins.
- **`name`**: A name scope for the associated operations (optional).

Returns:
A vector with the same dtype as `weights` or the given `dtype`. The bin values.

tf.math.ceil

- Contents
- Aliases:
- Used in the tutorials:

Returns element-wise smallest integer not less than x.

Aliases:

- `tf.compat.v1.ceil`
- `tf.compat.v1.math.ceil`
- `tf.compat.v2.math.ceil`
- `tf.math.ceil`

```
tf.math.ceil(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- [Load images with tf.data]()

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.confusion_matrix

- Contents
- Aliases:

Computes the confusion matrix from predictions and labels.

Aliases:

- `tf.compat.v2.math.confusion_matrix`
- `tf.math.confusion_matrix`

```
tf.math.confusion_matrix(
    labels,
    predictions,
    num_classes=None,
    weights=None,
    dtype=tf.dtypes.int32,
    name=None
)
```

Defined in `python/ops/confusion_matrix.py`.

The matrix columns represent the prediction labels and the rows represent the real labels. The confusion matrix is always a 2-D array of shape `[n, n]`, where `n` is the number of valid labels for a given classification task. Both prediction and labels must be 1-D arrays of the same shape in order for this function to work.

If `num_classes` is `None`, then `num_classes` will be set to one plus the maximum value in either predictions or labels. Class labels are expected to start at 0. For example, if `num_classes` is 3, then the possible labels would be `[0, 1, 2]`.
If `weights` is not `None`, then each prediction contributes its corresponding weight to the total value of the confusion matrix cell.

For example:

```
tf.math.confusion_matrix([1, 2, 4], [2, 2, 4]) ==>
    [[0 0 0 0 0]
     [0 0 1 0 0]
     [0 0 1 0 0]
     [0 0 0 0 0]
     [0 0 0 0 1]]
```

Note that the possible labels are assumed to be `[0, 1, 2, 3, 4]`, resulting in a 5x5 confusion matrix.

Args:

- **`labels`**: 1-D `Tensor` of real labels for the classification task.
- **`predictions`**: 1-D `Tensor` of predictions for a given classification.
- **`num_classes`**: The possible number of labels the classification task can have. If this value is not provided, it will be calculated using both predictions and labels array.
- **`weights`**: An optional `Tensor` whose shape matches `predictions`.
- **`dtype`**: Data type of the confusion matrix.
- **`name`**: Scope name.

Returns:

A `Tensor` of type `dtype` with shape `[n, n]` representing the confusion matrix, where `n` is the number of possible labels in the classification task.

Raises:

- **`ValueError`**: If both predictions and labels are not 1-D vectors and have mismatched shapes, or if `weights` is not `None` and its shape doesn't match `predictions`.

tf.math.conj

- Contents
- Aliases:

Returns the complex conjugate of a complex number.

Aliases:

- `tf.compat.v1.conj`
- `tf.compat.v1.math.conj`
- `tf.compat.v2.math.conj`
- `tf.math.conj`

```
tf.math.conj(
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Given a tensor `input` of complex numbers, this operation returns a tensor of complex numbers that are the complex conjugate of each element in `input`. The complex numbers in `input` must be of the form a+bj, where *a* is the real part and *b* is the imaginary part.

The complex conjugate returned by this operation is of the form a−bj.

For example:

tensor 'input' is [-2.25 + 4.75j, 3.25 + 5.75j]

tf.math.conj(input) ==> [-2.25 - 4.75j, 3.25 - 5.75j]

If `x` is real, it is returned unchanged.

Args:

- **`x`**: `Tensor` to conjugate. Must have numeric or variant type.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` that is the conjugate of `x` (with the same type).

Raises:

- **`TypeError`**: If `x` is not a numeric tensor.

tf.math.cos

- Contents
- Aliases:

Computes cos of x element-wise.

Aliases:

- `tf.compat.v1.cos`
- `tf.compat.v1.math.cos`
- `tf.compat.v2.cos`
- `tf.compat.v2.math.cos`
- `tf.cos`
- `tf.math.cos`

```
tf.math.cos(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.cosh

- Contents
- Aliases:

Computes hyperbolic cosine of x element-wise.

Aliases:

- `tf.compat.v1.cosh`
- `tf.compat.v1.math.cosh`
- `tf.compat.v2.cosh`
- `tf.compat.v2.math.cosh`

- `tf.cosh`
- `tf.math.cosh`

```
tf.math.cosh(
    x,
    name=None
)

```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.count_nonzero

- Contents
- Aliases:

Computes number of nonzero elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.count_nonzero`
- `tf.math.count_nonzero`

```
tf.math.count_nonzero(
    input,
    axis=None,
    keepdims=None,
    dtype=tf.dtypes.int64,
    name=None
)

```

Defined in `python/ops/math_ops.py`.

Reduces `input` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.

If `axis` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

NOTE Floating point comparison to zero is done by exact floating point equality check. Small values are **not** rounded to zero for purposes of the nonzero check.

For example:

```
x = tf.constant([[0, 1, 0], [1, 1, 0]])
tf.math.count_nonzero(x)  # 3
tf.math.count_nonzero(x, 0)  # [1, 2, 0]
tf.math.count_nonzero(x, 1)  # [1, 2]
tf.math.count_nonzero(x, 1, keepdims=True)  # [[1], [2]]
tf.math.count_nonzero(x, [0, 1])  # 3
```

NOTE Strings are compared against zero-length empty string `""`. Any string with a size greater than zero is already considered as nonzero.

For example:

```
x = tf.constant(["", "a", "  ", "b", ""])
tf.math.count_nonzero(x) # 3, with "a", "  ", and "b" as nonzero strings.
```

Args:

- **`input`**: The tensor to reduce. Should be of numeric type, `bool`, or `string`.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input), rank(input))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`dtype`**: The output dtype; defaults to `tf.int64`.
- **`name`**: A name for the operation (optional).

Returns:

The reduced tensor (number of nonzero values).

tf.math.cumprod

- Contents
- Aliases:

Compute the cumulative product of the tensor `x` along `axis`.

Aliases:

- `tf.compat.v1.cumprod`
- `tf.compat.v1.math.cumprod`
- `tf.compat.v2.math.cumprod`
- `tf.math.cumprod`

```
tf.math.cumprod(
    x,
    axis=0,
    exclusive=False,
    reverse=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

By default, this op performs an inclusive cumprod, which means that the first element of the input is identical to the first element of the output:

```
tf.math.cumprod([a, b, c])  # [a, a * b, a * b * c]
```

By setting the `exclusive` kwarg to `True`, an exclusive cumprod is performed instead:

```
tf.math.cumprod([a, b, c], exclusive=True)  # [1, a, a * b]
```

By setting the `reverse` kwarg to `True`, the cumprod is performed in the opposite direction:

```
tf.math.cumprod([a, b, c], reverse=True)  # [a * b * c, b * c, c]
```

This is more efficient than using separate `tf.reverse` ops. The `reverse` and `exclusive` kwargs can also be combined:

```
tf.math.cumprod([a, b, c], exclusive=True, reverse=True)  # [b * c, c, 1]
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- **`axis`**: A `Tensor` of type `int32` (default: 0). Must be in the range `[-rank(x), rank(x))`.
- **`exclusive`**: If `True`, perform exclusive cumprod.
- **`reverse`**: A `bool` (default: False).
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.cumsum

- Contents
- Aliases:

Compute the cumulative sum of the tensor `x` along `axis`.

Aliases:

- `tf.compat.v1.cumsum`
- `tf.compat.v1.math.cumsum`
- `tf.compat.v2.cumsum`
- `tf.compat.v2.math.cumsum`
- `tf.cumsum`
- `tf.math.cumsum`

```
tf.math.cumsum(
    x,
    axis=0,
    exclusive=False,
    reverse=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

By default, this op performs an inclusive cumsum, which means that the first element of the input is identical to the first element of the output:

```
tf.cumsum([a, b, c])  # [a, a + b, a + b + c]
```

By setting the `exclusive` kwarg to `True`, an exclusive cumsum is performed instead:

```
tf.cumsum([a, b, c], exclusive=True)  # [0, a, a + b]
```

By setting the `reverse` kwarg to `True`, the cumsum is performed in the opposite direction:

```
tf.cumsum([a, b, c], reverse=True)  # [a + b + c, b + c, c]
```

This is more efficient than using separate `tf.reverse` ops.

The `reverse` and `exclusive` kwargs can also be combined:

```
tf.cumsum([a, b, c], exclusive=True, reverse=True)  # [b + c, c, 0]
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- **`axis`**: A `Tensor` of type `int32` (default: 0). Must be in the range `[-rank(x), rank(x))`.
- **`exclusive`**: If `True`, perform exclusive cumsum.
- **`reverse`**: A `bool` (default: False).
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.digamma

- Contents
- Aliases:

Computes Psi, the derivative of Lgamma (the log of the absolute value of

Aliases:

- `tf.compat.v1.digamma`
- `tf.compat.v1.math.digamma`
- `tf.compat.v2.math.digamma`
- `tf.math.digamma`

```
tf.math.digamma(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

`Gamma(x)`), element-wise.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.divide

- Contents
- Aliases:

Computes Python style division of `x` by `y`.

Aliases:

- `tf.compat.v1.divide`
- `tf.compat.v1.math.divide`
- `tf.compat.v2.divide`
- `tf.compat.v2.math.divide`
- `tf.divide`
- `tf.math.divide`

```
tf.math.divide(
    x,
    y,
```

```
    name=None
)
```

Defined in `python/ops/math_ops.py`.

tf.math.divide_no_nan

- Contents
- Aliases:

Computes an unsafe divide which returns 0 if the y is zero.

Aliases:

- `tf.compat.v1.div_no_nan`
- `tf.compat.v1.math.divide_no_nan`
- `tf.compat.v2.math.divide_no_nan`
- `tf.math.divide_no_nan`

```
tf.math.divide_no_nan(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- **`y`**: A `Tensor` whose dtype is compatible with `x`.
- **`name`**: A name for the operation (optional).

Returns:

The element-wise value of the x divided by y.

tf.math.equal

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

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

Aliases:

- `tf.compat.v1.equal`
- `tf.compat.v1.math.equal`
- `tf.compat.v2.equal`
- `tf.compat.v2.math.equal`
- `tf.equal`
- `tf.math.equal`

```
tf.math.equal(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the guide:

- Training and Evaluation with TensorFlow Keras
- tf.function and AutoGraph in TensorFlow 2.0

Used in the tutorials:

- Image Captioning with Attention
- Load CSV with tf.data
- Neural Machine Translation with Attention
- Transformer model for language understanding
- tf.function

NOTE: `math.equal` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `quint8`, `qint8`, `qint32`, `string`, `bool`, `complex128`.
- **`y`**: A `Tensor`. Must have the same type as `x`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

tf.math.erf

- Contents
- Aliases:

Computes the Gauss error function of `x` element-wise.

Aliases:

- `tf.compat.v1.erf`
- `tf.compat.v1.math.erf`
- `tf.compat.v2.math.erf`
- `tf.math.erf`

```
tf.math.erf(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.erf(x.values, ...), x.dense_shape)`

tf.math.erfc

- Contents
- Aliases:

Computes the complementary error function of `x` element-wise.

Aliases:

- `tf.compat.v1.erfc`
- `tf.compat.v1.math.erfc`
- `tf.compat.v2.math.erfc`
- `tf.math.erfc`

```
tf.math.erfc(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.exp

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes exponential of x element-wise. y=ex.

Aliases:

- `tf.compat.v1.exp`
- `tf.compat.v1.math.exp`
- `tf.compat.v2.exp`
- `tf.compat.v2.math.exp`
- `tf.exp`
- `tf.math.exp`

```
tf.math.exp(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the guide:

- Eager essentials
- Writing layers and models with TensorFlow Keras

Used in the tutorials:

- Convolutional Variational Autoencoder

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.expm1

- Contents
- Aliases:

Computes exponential of x - 1 element-wise.

Aliases:

- `tf.compat.v1.expm1`
- `tf.compat.v1.math.expm1`
- `tf.compat.v2.math.expm1`
- `tf.math.expm1`

```
tf.math.expm1(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
I.e., $y=(\exp x)-1$.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.floor

- Contents
- Aliases:

Returns element-wise largest integer not greater than x.

Aliases:

- `tf.compat.v1.floor`
- `tf.compat.v1.math.floor`
- `tf.compat.v2.floor`
- `tf.compat.v2.math.floor`
- `tf.floor`
- `tf.math.floor`

```
tf.math.floor(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

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

tf.math.floordiv

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__floordiv__`
- `tf.compat.v1.RaggedTensor.__floordiv__`
- `tf.compat.v1.floordiv`
- `tf.compat.v1.math.floordiv`
- `tf.compat.v2.RaggedTensor.__floordiv__`
- `tf.compat.v2.math.floordiv`
- `tf.math.floordiv`

```
tf.math.floordiv(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
The same as `tf.compat.v1.div(x,y)` for integers, but uses `tf.floor(tf.compat.v1.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`.
`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.

Raises:

- **`TypeError`**: If the inputs are complex.

tf.math.floormod

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__mod__`
- `tf.compat.v1.RaggedTensor.__mod__`
- `tf.compat.v1.floormod`
- `tf.compat.v1.math.floormod`
- `tf.compat.v1.math.mod`
- `tf.compat.v1.mod`

- `tf.compat.v2.RaggedTensor.__mod__`
- `tf.compat.v2.math.floormod`
- `tf.compat.v2.math.mod`
- `tf.math.floormod`
- `tf.math.mod`

```
tf.math.floormod(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
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: `math.floormod` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `int32`, `int64`, `bfloat16`, `half`, `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`.

tf.math.greater

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__gt__`
- `tf.Tensor.__gt__`
- `tf.compat.v1.RaggedTensor.__gt__`
- `tf.compat.v1.Tensor.__gt__`
- `tf.compat.v1.greater`
- `tf.compat.v1.math.greater`
- `tf.compat.v2.RaggedTensor.__gt__`
- `tf.compat.v2.Tensor.__gt__`
- `tf.compat.v2.greater`
- `tf.compat.v2.math.greater`
- `tf.greater`
- `tf.math.greater`

```
tf.math.greater(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
NOTE: `math.greater` supports broadcasting. More about broadcasting here

Args:

- **x**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **y**: A `Tensor`. Must have the same type as `x`.
- **name**: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

tf.math.greater_equal

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__ge__`
- `tf.Tensor.__ge__`
- `tf.compat.v1.RaggedTensor.__ge__`
- `tf.compat.v1.Tensor.__ge__`
- `tf.compat.v1.greater_equal`
- `tf.compat.v1.math.greater_equal`
- `tf.compat.v2.RaggedTensor.__ge__`
- `tf.compat.v2.Tensor.__ge__`
- `tf.compat.v2.greater_equal`
- `tf.compat.v2.math.greater_equal`
- `tf.greater_equal`
- `tf.math.greater_equal`

```
tf.math.greater_equal(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

NOTE: `math.greater_equal` supports broadcasting. More about broadcasting here

Args:

- **x**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **y**: A `Tensor`. Must have the same type as `x`.
- **name**: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

tf.math.igamma

- Contents
- Aliases:

Compute the lower regularized incomplete Gamma function `P(a, x)`.

Aliases:

- `tf.compat.v1.igamma`
- `tf.compat.v1.math.igamma`
- `tf.compat.v2.math.igamma`
- `tf.math.igamma`

```
tf.math.igamma(
    a,
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

The lower regularized incomplete Gamma function is defined as:

P(a,x)=gamma(a,x)/Gamma(a)=1−Q(a,x)

where

gamma(a,x)=int0xta−1exp(−t)dt

is the lower incomplete Gamma function.

Note, above `Q(a, x)` (`Igammac`) is the upper regularized complete Gamma function.

Args:

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

Returns:

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

tf.math.igammac

- Contents
- Aliases:

Compute the upper regularized incomplete Gamma function `Q(a, x)`.

Aliases:

- `tf.compat.v1.igammac`
- `tf.compat.v1.math.igammac`
- `tf.compat.v2.math.igammac`
- `tf.math.igammac`

```
tf.math.igammac(
    a,
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

The upper regularized incomplete Gamma function is defined as:

Q(a,x)=Gamma(a,x)/Gamma(a)=1−P(a,x)

where

Gamma(a,x)=intx∞ta−1exp(−t)dt

is the upper incomplete Gama function.

Note, above `P(a, x)` (`Igamma`) is the lower regularized complete Gamma function.

Args:

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

Returns:

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

tf.math.imag

- Contents
- Aliases:

Returns the imaginary part of a complex (or real) tensor.

Aliases:

- `tf.compat.v1.imag`
- `tf.compat.v1.math.imag`
- `tf.compat.v2.math.imag`
- `tf.math.imag`

```
tf.math.imag(
    input,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Given a tensor `input`, this operation returns a tensor of type `float` that is the imaginary part of each element in `input` considered as a complex number. If `input` is real, a tensor of all zeros is returned.

For example:

```
x = tf.constant([-2.25 + 4.75j, 3.25 + 5.75j])
tf.math.imag(x)  # [4.75, 5.75]
```

Args:

- **input**: A `Tensor`. Must be one of the following types: `float`, `double`, `complex64`, `complex128`.
- **name**: A name for the operation (optional).

Returns:

A `Tensor` of type `float32` or `float64`.

tf.math.invert_permutation

- Contents
- Aliases:

Computes the inverse permutation of a tensor.

Aliases:

- `tf.compat.v1.invert_permutation`
- `tf.compat.v1.math.invert_permutation`
- `tf.compat.v2.math.invert_permutation`
- `tf.math.invert_permutation`

```
tf.math.invert_permutation(
    x,
    name=None
```

```
)
```

Defined in generated file: `python/ops/gen_array_ops.py`.
This operation computes the inverse of an index permutation. It takes a 1-D integer tensor `x`, which represents the indices of a zero-based array, and swaps each value with its index position. In other words, for an output tensor `y` and an input tensor `x`, this operation computes the following:
`y[x[i]] = i for i in [0, 1, ..., len(x) - 1]`
The values must include 0. There can be no duplicate values or negative values.

For example:

```
# tensor `x` is [3, 4, 0, 2, 1]
invert_permutation(x) ==> [2, 4, 3, 0, 1]
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. 1-D.
- **`name`**: A name for the operation (optional).

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

tf.math.in_top_k

- Contents
- Aliases:

Says whether the targets are in the top `K` predictions.

Aliases:

- `tf.compat.v2.math.in_top_k`
- `tf.compat.v2.nn.in_top_k`
- `tf.math.in_top_k`
- `tf.nn.in_top_k`

```
tf.math.in_top_k(
    targets,
    predictions,
    k,
    name=None
)
```

Defined in `python/ops/nn_ops.py`.
This outputs a `batch_size` bool array, an entry `out[i]` is `true` if the prediction for the target class is finite (not inf, -inf, or nan) and among the top `k` predictions among all predictions for example `i`. Note that the behavior of `InTopK` differs from the `TopK` op in its handling of ties; if multiple classes have the same prediction value and straddle the top-`k` boundary, all of those classes are considered to be in the top `k`.
More formally, let
predictionsi be the predictions for all classes for example `i`, targetsi be the target class for example `i`, outi be the output for example `i`,

outi=predictionsi,targetsi∈TopKIncludingTies(predictionsi)

Args:

- **`predictions`**: A `Tensor` of type `float32`. A `batch_size` x `classes` tensor.
- **`targets`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A `batch_size` vector of class ids.

- **k**: An `int`. Number of top elements to look at for computing precision.
- **name**: A name for the operation (optional).

Returns:
A `Tensor` of type `bool`. Computed Precision at `k` as a `bool Tensor`.

tf.math.is_finite

- Contents
- Aliases:

Returns which elements of x are finite.

Aliases:

- `tf.compat.v1.debugging.is_finite`
- `tf.compat.v1.is_finite`
- `tf.compat.v1.math.is_finite`
- `tf.compat.v2.math.is_finite`
- `tf.math.is_finite`

```
tf.math.is_finite(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **x**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **name**: A name for the operation (optional).

Returns:
A `Tensor` of type `bool`.

Numpy Compatibility
Equivalent to np.isfinite

tf.math.is_inf

- Contents
- Aliases:

Returns which elements of x are Inf.

Aliases:

- `tf.compat.v1.debugging.is_inf`
- `tf.compat.v1.is_inf`
- `tf.compat.v1.math.is_inf`
- `tf.compat.v2.math.is_inf`
- `tf.math.is_inf`

```
tf.math.is_inf(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **x**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.

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

Returns:
A `Tensor` of type `bool`.

Numpy Compatibility
Equivalent to np.isinf

tf.math.is_nan

- Contents
- Aliases:

Returns which elements of x are NaN.

Aliases:

- `tf.compat.v1.debugging.is_nan`
- `tf.compat.v1.is_nan`
- `tf.compat.v1.math.is_nan`
- `tf.compat.v2.math.is_nan`
- `tf.math.is_nan`

```
tf.math.is_nan(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- `x`: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- `name`: A name for the operation (optional).

Returns:
A `Tensor` of type `bool`.

Numpy Compatibility
Equivalent to np.isnan

tf.math.is_non_decreasing

- Contents
- Aliases:

Returns `True` if `x` is non-decreasing.

Aliases:

- `tf.compat.v1.debugging.is_non_decreasing`
- `tf.compat.v1.is_non_decreasing`
- `tf.compat.v1.math.is_non_decreasing`
- `tf.compat.v2.math.is_non_decreasing`
- `tf.math.is_non_decreasing`

```
tf.math.is_non_decreasing(
    x,
    name=None
)
```

Defined in `python/ops/check_ops.py`.

Elements of `x` are compared in row-major order. The tensor `[x[0],...]` is non-decreasing if for every adjacent pair we have `x[i] <= x[i+1]`. If `x` has less than two elements, it is trivially non-decreasing.
See also: `is_strictly_increasing`

Args:

- **`x`**: Numeric `Tensor`.
- **`name`**: A name for this operation (optional). Defaults to "is_non_decreasing"

Returns:
Boolean `Tensor`, equal to `True` iff `x` is non-decreasing.

Raises:

- **`TypeError`**: if `x` is not a numeric tensor.

tf.math.is_strictly_increasing

- Contents
- Aliases:

Returns `True` if `x` is strictly increasing.

Aliases:

- `tf.compat.v1.debugging.is_strictly_increasing`
- `tf.compat.v1.is_strictly_increasing`
- `tf.compat.v1.math.is_strictly_increasing`
- `tf.compat.v2.math.is_strictly_increasing`
- `tf.math.is_strictly_increasing`

```
tf.math.is_strictly_increasing(
    x,
    name=None
)
```

Defined in `python/ops/check_ops.py`.
Elements of `x` are compared in row-major order. The tensor `[x[0],...]` is strictly increasing if for every adjacent pair we have `x[i] < x[i+1]`. If `x` has less than two elements, it is trivially strictly increasing.
See also: `is_non_decreasing`

Args:

- **`x`**: Numeric `Tensor`.
- **`name`**: A name for this operation (optional). Defaults to "is_strictly_increasing"

Returns:
Boolean `Tensor`, equal to `True` iff `x` is strictly increasing.

Raises:

- **`TypeError`**: if `x` is not a numeric tensor.

tf.math.l2_normalize

- Contents
- Aliases:

Normalizes along dimension `axis` using an L2 norm.

Aliases:

- `tf.compat.v2.linalg.l2_normalize`
- `tf.compat.v2.math.l2_normalize`

- `tf.compat.v2.nn.l2_normalize`
- `tf.linalg.l2_normalize`
- `tf.math.l2_normalize`
- `tf.nn.l2_normalize`

```
tf.math.l2_normalize(
    x,
    axis=None,
    epsilon=1e-12,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

For a 1-D tensor with `axis = 0`, computes

```
output = x / sqrt(max(sum(x**2), epsilon))
```

For `x` with more dimensions, independently normalizes each 1-D slice along dimension `axis`.

Args:

- **`x`**: A `Tensor`.
- **`axis`**: Dimension along which to normalize. A scalar or a vector of integers.
- **`epsilon`**: A lower bound value for the norm. Will use `sqrt(epsilon)` as the divisor if `norm < sqrt(epsilon)`.
- **`name`**: A name for this operation (optional).

Returns:

A `Tensor` with the same shape as `x`.

tf.math.lbeta

- Contents
- Aliases:

Computes ln(|Beta(x)|), reducing along the last dimension.

Aliases:

- `tf.compat.v1.lbeta`
- `tf.compat.v1.math.lbeta`
- `tf.compat.v2.math.lbeta`
- `tf.math.lbeta`

```
tf.math.lbeta(
    x,
    name=None
)
```

Defined in `python/ops/special_math_ops.py`.

Given one-dimensional `z = [z_0,...,z_{K-1}]`, we define

Beta(z)=∏jGamma(zj)/Gamma(∑jzj)

And for `n + 1` dimensional `x` with shape `[N1, ..., Nn, K]`, we define lbeta(x)[i1,...,in]=Log(|Beta(x[i1,...,in,:])|).

In other words, the last dimension is treated as the `z` vector.

Note that if `z = [u, v]`, then Beta(z)=int01tu−1(1−t)v−1dt, which defines the traditional bivariate beta function.

If the last dimension is empty, we follow the convention that the sum over the empty set is zero, and the product is one.

Args:

- **x**: A rank `n + 1` `Tensor`, `n >= 0` with type `float`, or `double`.
- **name**: A name for the operation (optional).

Returns:

The logarithm of |Beta(x)| reducing along the last dimension.

tf.math.less

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__lt__`
- `tf.Tensor.__lt__`
- `tf.compat.v1.RaggedTensor.__lt__`
- `tf.compat.v1.Tensor.__lt__`
- `tf.compat.v1.less`
- `tf.compat.v1.math.less`
- `tf.compat.v2.RaggedTensor.__lt__`
- `tf.compat.v2.Tensor.__lt__`
- `tf.compat.v2.less`
- `tf.compat.v2.math.less`
- `tf.less`
- `tf.math.less`

```
tf.math.less(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

NOTE: `math.less` supports broadcasting. More about broadcasting here

Args:

- **x**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **y**: A `Tensor`. Must have the same type as `x`.
- **name**: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

tf.math.less_equal

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__le__`
- `tf.Tensor.__le__`
- `tf.compat.v1.RaggedTensor.__le__`
- `tf.compat.v1.Tensor.__le__`
- `tf.compat.v1.less_equal`
- `tf.compat.v1.math.less_equal`
- `tf.compat.v2.RaggedTensor.__le__`
- `tf.compat.v2.Tensor.__le__`
- `tf.compat.v2.less_equal`
- `tf.compat.v2.math.less_equal`
- `tf.less_equal`
- `tf.math.less_equal`

```
tf.math.less_equal(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

NOTE: `math.less_equal` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **`y`**: A `Tensor`. Must have the same type as `x`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor` of type `bool`.

tf.math.lgamma

- Contents
- Aliases:

Computes the log of the absolute value of `Gamma(x)` element-wise.

Aliases:

- `tf.compat.v1.lgamma`
- `tf.compat.v1.math.lgamma`
- `tf.compat.v2.math.lgamma`
- `tf.math.lgamma`

```
tf.math.lgamma(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.

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

Returns:

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

tf.math.log

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes natural logarithm of x element-wise.

Aliases:

- `tf.compat.v1.log`
- `tf.compat.v1.math.log`
- `tf.compat.v2.math.log`
- `tf.math.log`

```
tf.math.log(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the guide:

- Eager essentials

Used in the tutorials:

- Convolutional Variational Autoencoder

I.e., $y=\log_e x$.

Args:

- `x`: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name`: A name for the operation (optional).

Returns:

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

tf.math.log1p

- Contents
- Aliases:

Computes natural logarithm of (1 + x) element-wise.

Aliases:

- `tf.compat.v1.log1p`
- `tf.compat.v1.math.log1p`
- `tf.compat.v2.math.log1p`
- `tf.math.log1p`

```
tf.math.log1p(
    x,
    name=None
```

```
)

```

Defined in generated file: `python/ops/gen_math_ops.py`.
I.e., y=loge(1+x).

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.logical_and

- Contents
- Aliases:
- Used in the tutorials:

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

Aliases:

- `tf.RaggedTensor.__and__`
- `tf.compat.v1.RaggedTensor.__and__`
- `tf.compat.v1.logical_and`
- `tf.compat.v1.math.logical_and`
- `tf.compat.v2.RaggedTensor.__and__`
- `tf.compat.v2.logical_and`
- `tf.compat.v2.math.logical_and`
- `tf.logical_and`
- `tf.math.logical_and`

```
tf.math.logical_and(
    x,
    y,
    name=None
)

```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- [Transformer model for language understanding]()

NOTE: `math.logical_and` 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`.

tf.math.logical_not

- Contents
- Aliases:
- Used in the tutorials:

Returns the truth value of NOT x element-wise.

Aliases:

- `tf.RaggedTensor.__invert__`
- `tf.Tensor.__invert__`
- `tf.compat.v1.RaggedTensor.__invert__`
- `tf.compat.v1.Tensor.__invert__`
- `tf.compat.v1.logical_not`
- `tf.compat.v1.math.logical_not`
- `tf.compat.v2.RaggedTensor.__invert__`
- `tf.compat.v2.Tensor.__invert__`
- `tf.compat.v2.logical_not`
- `tf.compat.v2.math.logical_not`
- `tf.logical_not`
- `tf.math.logical_not`

```
tf.math.logical_not(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Image Captioning with Attention
- Neural Machine Translation with Attention
- Transformer model for language understanding

Args:

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

Returns:

A `Tensor` of type `bool`.

tf.math.logical_or

- Contents
- Aliases:

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

Aliases:

- `tf.RaggedTensor.__or__`
- `tf.compat.v1.RaggedTensor.__or__`
- `tf.compat.v1.logical_or`
- `tf.compat.v1.math.logical_or`
- `tf.compat.v2.RaggedTensor.__or__`
- `tf.compat.v2.logical_or`
- `tf.compat.v2.math.logical_or`
- `tf.logical_or`
- `tf.math.logical_or`

```
tf.math.logical_or(
    x,
    y,
```

```
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

NOTE: `math.logical_or` 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`.

tf.math.logical_xor

- Contents
- Aliases:

Logical XOR function.

Aliases:

- `tf.RaggedTensor.__xor__`
- `tf.compat.v1.RaggedTensor.__xor__`
- `tf.compat.v1.logical_xor`
- `tf.compat.v1.math.logical_xor`
- `tf.compat.v2.RaggedTensor.__xor__`
- `tf.compat.v2.math.logical_xor`
- `tf.math.logical_xor`

```
tf.math.logical_xor(
    x,
    y,
    name='LogicalXor'
)
```

Defined in `python/ops/math_ops.py`.

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

Inputs are tensor and if the tensors contains more than one element, an element-wise logical XOR is computed.

Usage:

```
x = tf.constant([False, False, True, True], dtype = tf.bool)
y = tf.constant([False, True, False, True], dtype = tf.bool)
z = tf.logical_xor(x, y, name="LogicalXor")
#  here z = [False  True  True False]
```

Args:

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

Returns:

A `Tensor` of type bool with the same size as that of x or y.

tf.math.log_sigmoid

- Contents
- Aliases:

Computes log sigmoid of `x` element-wise.

Aliases:

- `tf.compat.v1.log_sigmoid`
- `tf.compat.v1.math.log_sigmoid`
- `tf.compat.v2.math.log_sigmoid`
- `tf.math.log_sigmoid`

```
tf.math.log_sigmoid(
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Specifically, `y = log(1 / (1 + exp(-x)))`. For numerical stability, we use `y = -tf.nn.softplus(-x)`.

Args:

- **`x`**: A Tensor with type `float32` or `float64`.
- **`name`**: A name for the operation (optional).

Returns:

A Tensor with the same type as `x`.

tf.math.maximum

- Contents
- Aliases:
- Used in the tutorials:

Returns the max of x and y (i.e. x > y ? x : y) element-wise.

Aliases:

- `tf.compat.v1.math.maximum`
- `tf.compat.v1.maximum`
- `tf.compat.v2.math.maximum`
- `tf.compat.v2.maximum`
- `tf.math.maximum`
- `tf.maximum`

```
tf.math.maximum(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Transformer model for language understanding

NOTE: `math.maximum` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`.
- **`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`.

tf.math.minimum

- Contents
- Aliases:
- Used in the tutorials:

Returns the min of x and y (i.e. x < y ? x : y) element-wise.

Aliases:

- `tf.compat.v1.math.minimum`
- `tf.compat.v1.minimum`
- `tf.compat.v2.math.minimum`
- `tf.compat.v2.minimum`
- `tf.math.minimum`
- `tf.minimum`

```
tf.math.minimum(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Transformer model for language understanding

NOTE: `math.minimum` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`.
- **`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`.

tf.math.multiply

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Returns x * y element-wise.

Aliases:

- `tf.RaggedTensor.__mul__`
- `tf.compat.v1.RaggedTensor.__mul__`
- `tf.compat.v1.math.multiply`
- `tf.compat.v1.multiply`
- `tf.compat.v2.RaggedTensor.__mul__`
- `tf.compat.v2.math.multiply`
- `tf.compat.v2.multiply`

- `tf.math.multiply`
- `tf.multiply`

```
tf.math.multiply(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Training and Evaluation with TensorFlow Keras

Used in the tutorials:

- Automatic differentiation and gradient tape
- Tensors and Operations

NOTE: `<a href="../../tf/math/multiply"><code>tf.multiply</code></a>` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `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`.

tf.math.multiply_no_nan

- Contents
- Aliases:

Computes the product of x and y and returns 0 if the y is zero, even if x is NaN or infinite.

Aliases:

- `tf.compat.v1.math.multiply_no_nan`
- `tf.compat.v2.math.multiply_no_nan`
- `tf.math.multiply_no_nan`

```
tf.math.multiply_no_nan(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- **`y`**: A `Tensor` whose dtype is compatible with `x`.
- **`name`**: A name for the operation (optional).

Returns:

The element-wise value of the x times y.

tf.math.negative

- Contents
- Aliases:

Computes numerical negative value element-wise.

Aliases:

- `tf.RaggedTensor.__neg__`
- `tf.Tensor.__neg__`
- `tf.compat.v1.RaggedTensor.__neg__`
- `tf.compat.v1.Tensor.__neg__`
- `tf.compat.v1.math.negative`
- `tf.compat.v1.negative`
- `tf.compat.v2.RaggedTensor.__neg__`
- `tf.compat.v2.Tensor.__neg__`
- `tf.compat.v2.math.negative`
- `tf.compat.v2.negative`
- `tf.math.negative`
- `tf.negative`

```
tf.math.negative(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

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

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.negative(x.values, ...), x.dense_shape)`

tf.math.nextafter

- Contents
- Aliases:

Returns the next representable value of `x1` in the direction of `x2`, element-wise.

Aliases:

- `tf.compat.v1.math.nextafter`
- `tf.compat.v2.math.nextafter`
- `tf.math.nextafter`

```
tf.math.nextafter(
    x1,
    x2,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
This operation returns the same result as the C++ std::nextafter function.
It can also return a subnormal number.

Args:

- **`x1`**: A `Tensor`. Must be one of the following types: `float64`, `float32`.
- **`x2`**: A `Tensor`. Must have the same type as `x1`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has the same type as `x1`.

Cpp Compatibility
Equivalent to C++ std::nextafter function.

tf.math.not_equal

- Contents
- Aliases:
- Used in the tutorials:

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

Aliases:

- `tf.compat.v1.math.not_equal`
- `tf.compat.v1.not_equal`
- `tf.compat.v2.math.not_equal`
- `tf.compat.v2.not_equal`
- `tf.math.not_equal`
- `tf.not_equal`

```
tf.math.not_equal(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Unicode strings

NOTE: `math.not_equal` supports broadcasting. More about broadcasting here

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `quint8`, `qint8`, `qint32`, `string`, `bool`, `complex128`.
- **`y`**: A `Tensor`. Must have the same type as `x`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of type `bool`.

tf.math.polygamma

- Contents
- Aliases:

Compute the polygamma function ψ(n)(x).

Aliases:

- `tf.compat.v1.math.polygamma`
- `tf.compat.v1.polygamma`
- `tf.compat.v2.math.polygamma`
- `tf.math.polygamma`

```
tf.math.polygamma(
    a,
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
The polygamma function is defined as:
ψ(a)(x)=dadxaψ(x)
where ψ(x) is the digamma function. The polygamma function is defined only for non-negative integer orders \a\.

Args:

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

Returns:
A `Tensor`. Has the same type as `a`.

tf.math.polyval

- Contents
- Aliases:

Computes the elementwise value of a polynomial.

Aliases:

- `tf.compat.v1.math.polyval`
- `tf.compat.v2.math.polyval`
- `tf.math.polyval`

```
tf.math.polyval(
    coeffs,
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
If `x` is a tensor and `coeffs` is a list n + 1 tensors, this function returns the value of the n-th order polynomial
p(x) = coeffs[n-1] + coeffs[n-2] * x + ... + coeffs[0] * x**(n-1)
evaluated using Horner's method, i.e.
p(x) = coeffs[n-1] + x * (coeffs[n-2] + ... + x * (coeffs[1] + x * coeffs[0]))

Args:

- **`coeffs`**: A list of `Tensor` representing the coefficients of the polynomial.
- **`x`**: A `Tensor` representing the variable of the polynomial.
- **`name`**: A name for the operation (optional).

Returns:
A `tensor` of the shape as the expression p(x) with usual broadcasting rules for element-wise addition and multiplication applied.

Numpy Compatibility
Equivalent to numpy.polyval.

tf.math.pow

- Contents
- Aliases:

Computes the power of one value to another.

Aliases:

- `tf.RaggedTensor.__pow__`
- `tf.compat.v1.RaggedTensor.__pow__`
- `tf.compat.v1.math.pow`
- `tf.compat.v1.pow`
- `tf.compat.v2.RaggedTensor.__pow__`
- `tf.compat.v2.math.pow`
- `tf.compat.v2.pow`
- `tf.math.pow`
- `tf.pow`

```
tf.math.pow(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Given a tensor `x` and a tensor `y`, this operation computes xy 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 `float16`, `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- **`y`**: A `Tensor` of type `float16`, `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`.

tf.math.pow

- Contents
- Aliases:

Computes the power of one value to another.

Aliases:

- `tf.RaggedTensor.__pow__`
- `tf.compat.v1.RaggedTensor.__pow__`

- `tf.compat.v1.math.pow`
- `tf.compat.v1.pow`
- `tf.compat.v2.RaggedTensor.__pow__`
- `tf.compat.v2.math.pow`
- `tf.compat.v2.pow`
- `tf.math.pow`
- `tf.pow`

```
tf.math.pow(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Given a tensor `x` and a tensor `y`, this operation computes xy 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 `float16`, `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- **`y`**: A `Tensor` of type `float16`, `float32`, `float64`, `int32`, `int64`, `complex64`, or `complex128`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`.

tf.math.reciprocal

- Contents
- Aliases:

Computes the reciprocal of x element-wise.

Aliases:

- `tf.compat.v1.math.reciprocal`
- `tf.compat.v1.reciprocal`
- `tf.compat.v2.math.reciprocal`
- `tf.math.reciprocal`

```
tf.math.reciprocal(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
I.e., y=1/x.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.reduce_any

- Contents
- Aliases:

Computes the "logical or" of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_any`
- `tf.compat.v2.reduce_any`
- `tf.math.reduce_any`
- `tf.reduce_any`

```
tf.math.reduce_any(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[True,  True], [False, False]])
tf.reduce_any(x)  # True
tf.reduce_any(x, 0)  # [True, True]
tf.reduce_any(x, 1)  # [True, False]
```

Args:

- **`input_tensor`**: The boolean tensor to reduce.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor.

Numpy Compatibility
Equivalent to np.any

tf.math.reduce_euclidean_norm

- Contents
- Aliases:

Computes the Euclidean norm of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v1.math.reduce_euclidean_norm`

- `tf.compat.v2.math.reduce_euclidean_norm`
- `tf.math.reduce_euclidean_norm`

```
tf.math.reduce_euclidean_norm(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[1, 2, 3], [1, 1, 1]])
tf.reduce_euclidean_norm(x)  # sqrt(17)
tf.reduce_euclidean_norm(x, 0)  # [sqrt(2), sqrt(5), sqrt(10)]
tf.reduce_euclidean_norm(x, 1)  # [sqrt(14), sqrt(3)]
tf.reduce_euclidean_norm(x, 1, keepdims=True)  # [[sqrt(14)], [sqrt(3)]]
tf.reduce_euclidean_norm(x, [0, 1])  # sqrt(17)
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor, of the same dtype as the input_tensor.

tf.math.reduce_logsumexp

- Contents
- Aliases:

Computes log(sum(exp(elements across dimensions of a tensor))).

Aliases:

- `tf.compat.v2.math.reduce_logsumexp`
- `tf.compat.v2.reduce_logsumexp`
- `tf.math.reduce_logsumexp`
- `tf.reduce_logsumexp`

```
tf.math.reduce_logsumexp(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` has no entries, all dimensions are reduced, and a tensor with a single element is returned.
This function is more numerically stable than log(sum(exp(input))). It avoids overflows caused by taking the exp of large inputs and underflows caused by taking the log of small inputs.

For example:

```
x = tf.constant([[0., 0., 0.], [0., 0., 0.]])
tf.reduce_logsumexp(x)  # log(6)
tf.reduce_logsumexp(x, 0)  # [log(2), log(2), log(2)]
tf.reduce_logsumexp(x, 1)  # [log(3), log(3)]
tf.reduce_logsumexp(x, 1, keepdims=True)  # [[log(3)], [log(3)]]
tf.reduce_logsumexp(x, [0, 1])  # log(6)
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor.

tf.math.reduce_max

- Contents
- Aliases:

Computes the maximum of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_max`
- `tf.compat.v2.reduce_max`
- `tf.math.reduce_max`
- `tf.reduce_max`

```
tf.math.reduce_max(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

Args:

- **`input_tensor`**: The tensor to reduce. Should have real numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.

- `keepdims`: If true, retains reduced dimensions with length 1.
- `name`: A name for the operation (optional).

Returns:
The reduced tensor.

Numpy Compatibility
Equivalent to np.max

tf.math.reduce_mean

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes the mean of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_mean`
- `tf.compat.v2.reduce_mean`
- `tf.math.reduce_mean`
- `tf.reduce_mean`

```
tf.math.reduce_mean(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Eager essentials
- Ragged Tensors
- Training checkpoints
- Writing layers and models with TensorFlow Keras
- tf.function and AutoGraph in TensorFlow 2.0

Used in the tutorials:

- Convolutional Variational Autoencoder
- Custom training: basics
- Image Captioning with Attention
- Neural Machine Translation with Attention
- Neural style transfer
- Pix2Pix
- Text generation with an RNN
- Transformer model for language understanding

Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[1., 1.], [2., 2.]])
tf.reduce_mean(x)  # 1.5
tf.reduce_mean(x, 0)  # [1.5, 1.5]
tf.reduce_mean(x, 1)  # [1.,  2.]
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor.

Numpy Compatibility
Equivalent to np.mean
Please note that `np.mean` has a `dtype` parameter that could be used to specify the output type. By default this is `dtype=float64`. On the other hand, `tf.reduce_mean` has an aggressive type inference from `input_tensor`, for example:

```
x = tf.constant([1, 0, 1, 0])
tf.reduce_mean(x)  # 0
y = tf.constant([1., 0., 1., 0.])
tf.reduce_mean(y)  # 0.5
```

tf.math.reduce_min

- Contents
- Aliases:

Computes the minimum of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_min`
- `tf.compat.v2.reduce_min`
- `tf.math.reduce_min`
- `tf.reduce_min`

```
tf.math.reduce_min(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

Args:

- **`input_tensor`**: The tensor to reduce. Should have real numeric type.

- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor.

Numpy Compatibility
Equivalent to np.min

tf.math.reduce_prod

- Contents
- Aliases:

Computes the product of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_prod`
- `tf.compat.v2.reduce_prod`
- `tf.math.reduce_prod`
- `tf.reduce_prod`

```
tf.math.reduce_prod(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor.

Numpy Compatibility
Equivalent to np.prod

tf.math.reduce_std

- Contents
- Aliases:

Computes the standard deviation of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v1.math.reduce_std`

- `tf.compat.v2.math.reduce_std`
- `tf.math.reduce_std`

```
tf.math.reduce_std(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.

If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[1., 2.], [3., 4.]])
tf.reduce_std(x)  # 1.1180339887498949
tf.reduce_std(x, 0)  # [1., 1.]
tf.reduce_std(x, 1)  # [0.5,  0.5]
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name scope for the associated operations (optional).

Returns:

The reduced tensor, of the same dtype as the input_tensor.

Numpy Compatibility

Equivalent to np.std

Please note that `np.std` has a `dtype` parameter that could be used to specify the output type. By default this is `dtype=float64`. On the other hand, `tf.reduce_std` has an aggressive type inference from `input_tensor`,

tf.math.reduce_sum

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes the sum of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v2.math.reduce_sum`
- `tf.compat.v2.reduce_sum`
- `tf.math.reduce_sum`
- `tf.reduce_sum`

```
tf.math.reduce_sum(
    input_tensor,
    axis=None,
```

```
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the guide:

- Distributed training in TensorFlow
- Eager essentials
- Training and Evaluation with TensorFlow Keras
- Writing layers and models with TensorFlow Keras

Used in the tutorials:

- Automatic differentiation and gradient tape
- Convolutional Variational Autoencoder
- Image Captioning with Attention
- Multi-worker Training with Estimator
- Neural Machine Translation with Attention
- Tensors and Operations
- Unicode strings
- tf.function

Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.
If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[1, 1, 1], [1, 1, 1]])
tf.reduce_sum(x)  # 6
tf.reduce_sum(x, 0)  # [2, 2, 2]
tf.reduce_sum(x, 1)  # [3, 3]
tf.reduce_sum(x, 1, keepdims=True)  # [[3], [3]]
tf.reduce_sum(x, [0, 1])  # 6
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name for the operation (optional).

Returns:
The reduced tensor, of the same dtype as the input_tensor.

Numpy Compatibility
Equivalent to np.sum apart the fact that numpy upcast uint8 and int32 to int64 while tensorflow returns the same dtype as the input.

tf.math.reduce_variance

- Contents
- Aliases:

Computes the variance of elements across dimensions of a tensor.

Aliases:

- `tf.compat.v1.math.reduce_variance`
- `tf.compat.v2.math.reduce_variance`
- `tf.math.reduce_variance`

```
tf.math.reduce_variance(
    input_tensor,
    axis=None,
    keepdims=False,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Reduces `input_tensor` along the dimensions given in `axis`. Unless `keepdims` is true, the rank of the tensor is reduced by 1 for each entry in `axis`. If `keepdims` is true, the reduced dimensions are retained with length 1.

If `axis` is None, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
x = tf.constant([[1., 2.], [3., 4.]])
tf.reduce_variance(x)  # 1.25
tf.reduce_variance(x, 0)  # [1., 1.]
tf.reduce_variance(x, 1)  # [0.25,  0.25]
```

Args:

- **`input_tensor`**: The tensor to reduce. Should have numeric type.
- **`axis`**: The dimensions to reduce. If `None` (the default), reduces all dimensions. Must be in the range `[-rank(input_tensor), rank(input_tensor))`.
- **`keepdims`**: If true, retains reduced dimensions with length 1.
- **`name`**: A name scope for the associated operations (optional).

Returns:

The reduced tensor, of the same dtype as the input_tensor.

Numpy Compatibility

Equivalent to np.var

Please note that `np.var` has a `dtype` parameter that could be used to specify the output type. By default this is `dtype=float64`. On the other hand, `tf.reduce_variance` has an aggressive type inference from `input_tensor`,

tf.math.rint

- Contents
- Aliases:

Returns element-wise integer closest to x.

Aliases:

- `tf.compat.v1.math.rint`
- `tf.compat.v1.rint`
- `tf.compat.v2.math.rint`
- `tf.math.rint`

```
tf.math.rint(
    x,
```

```
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
If the result is midway between two representable values, the even representable is chosen. For example:

```
rint(-1.5) ==> -2.0
rint(0.5000001) ==> 1.0
rint([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0]) ==> [-2., -2., -0., 0., 2., 2., 2.]
```

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

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

tf.math.round

- Contents
- Aliases:

Rounds the values of a tensor to the nearest integer, element-wise.

Aliases:

- `tf.compat.v1.math.round`
- `tf.compat.v1.round`
- `tf.compat.v2.math.round`
- `tf.compat.v2.round`
- `tf.math.round`
- `tf.round`

```
tf.math.round(
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
Rounds half to even. Also known as bankers rounding. If you want to round according to the current system rounding mode use tf::cint. For example:

```
x = tf.constant([0.9, 2.5, 2.3, 1.5, -4.5])
tf.round(x)  # [ 1.0, 2.0, 2.0, 2.0, -4.0 ]
```

Args:

- **`x`**: A `Tensor` of type `float16`, `float32`, `float64`, `int32`, or `int64`.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor` of same shape and type as `x`.

tf.math.rsqrt

- Contents
- Aliases:
- Used in the tutorials:

Computes reciprocal of square root of x element-wise.

Aliases:

- `tf.compat.v1.math.rsqrt`
- `tf.compat.v1.rsqrt`
- `tf.compat.v2.math.rsqrt`
- `tf.math.rsqrt`

```
tf.math.rsqrt(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- [Transformer model for language understanding]()

I.e., y=1/x.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.scalar_mul

- Contents
- Aliases:

Multiplies a scalar times a `Tensor` or `IndexedSlices` object.

Aliases:

- `tf.compat.v2.math.scalar_mul`
- `tf.compat.v2.scalar_mul`
- `tf.math.scalar_mul`
- `tf.scalar_mul`

```
tf.math.scalar_mul(
    scalar,
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Intended for use in gradient code which might deal with `IndexedSlices` objects, which are easy to multiply by a scalar but more expensive to multiply with arbitrary tensors.

Args:

- **`scalar`**: A 0-D scalar `Tensor`. Must have known shape.
- **`x`**: A `Tensor` or `IndexedSlices` to be scaled.
- **`name`**: A name for the operation (optional).

Returns:

`scalar * x` of the same type (`Tensor` or `IndexedSlices`) as `x`.

Raises:

- **`ValueError`**: if scalar is not a 0-D `scalar`.

tf.math.segment_max

- Contents
- Aliases:

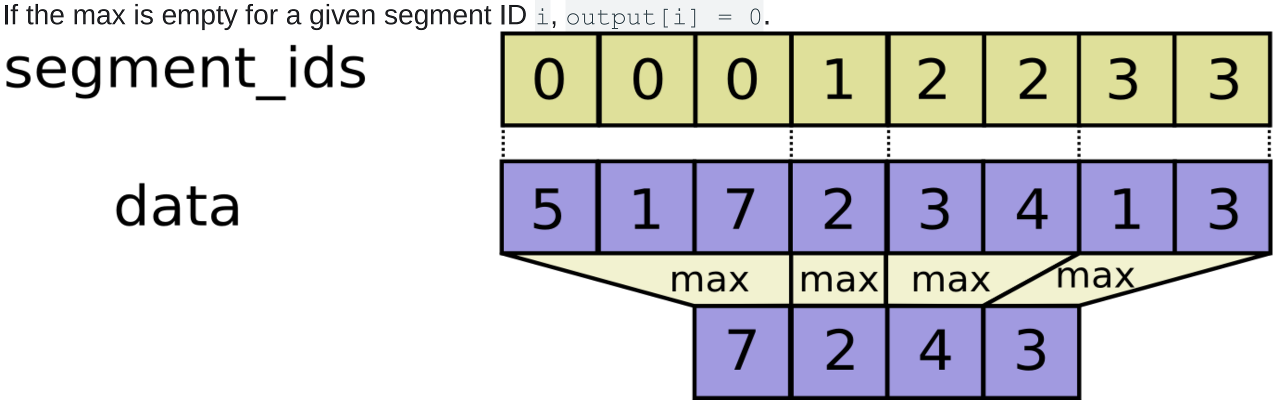
Computes the maximum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.segment_max`
- `tf.compat.v1.segment_max`
- `tf.compat.v2.math.segment_max`
- `tf.math.segment_max`

```
tf.math.segment_max(
    data,
    segment_ids,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
Computes a tensor such that outputi=maxj(dataj) where `max` is over `j` such that `segment_ids[j] == i`.
If the max is empty for a given segment ID `i`, `output[i] = 0`.



For example:

```
c = tf.constant([[1,2,3,4], [4, 3, 2, 1], [5,6,7,8]])
tf.segment_max(c, tf.constant([0, 0, 1]))
# ==> [[4, 3, 3, 4],
#      [5, 6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose size is equal to the size of `data`'s first dimension. Values should be sorted and can be repeated.
- **`name`**: A name for the operation (optional).

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

tf.math.segment_mean

- Contents
- Aliases:

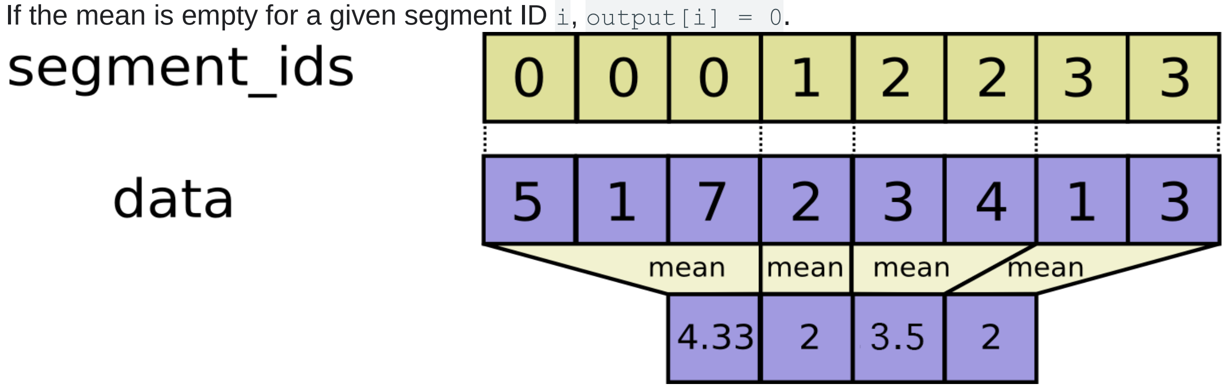
Computes the mean along segments of a tensor.

Aliases:

- `tf.compat.v1.math.segment_mean`
- `tf.compat.v1.segment_mean`
- `tf.compat.v2.math.segment_mean`
- `tf.math.segment_mean`

```
tf.math.segment_mean(
    data,
    segment_ids,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
Computes a tensor such that outputi=∑jdatajN where `mean` is over `j` such that `segment_ids[j] == i` and `N` is the total number of values summed.
If the mean is empty for a given segment ID `i`, `output[i] = 0`.



For example:

```
c = tf.constant([[1.0,2,3,4], [4, 3, 2, 1], [5,6,7,8]])
tf.segment_mean(c, tf.constant([0, 0, 1]))
# ==> [[2.5, 2.5, 2.5, 2.5],
#      [5, 6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose size is equal to the size of `data`'s first dimension. Values should be sorted and can be repeated.
- **`name`**: A name for the operation (optional).

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

tf.math.segment_min

- Contents
- Aliases:

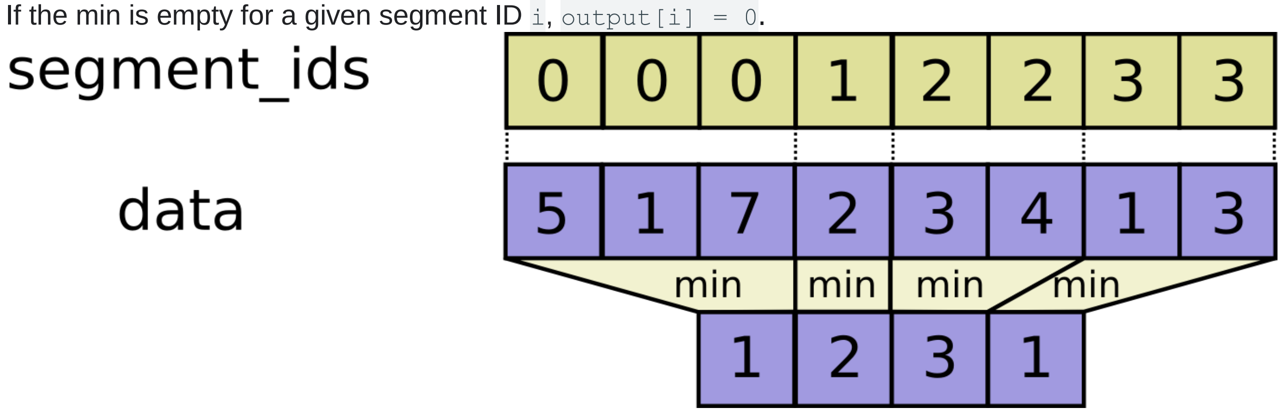
Computes the minimum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.segment_min`
- `tf.compat.v1.segment_min`
- `tf.compat.v2.math.segment_min`
- `tf.math.segment_min`

```
tf.math.segment_min(
    data,
    segment_ids,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
Computes a tensor such that outputi=minj(dataj) where `min` is over `j` such that `segment_ids[j] == i`.
If the min is empty for a given segment ID `i`, `output[i] = 0`.



For example:

```
c = tf.constant([[1,2,3,4], [4, 3, 2, 1], [5,6,7,8]])
tf.segment_min(c, tf.constant([0, 0, 1]))
# ==> [[1, 2, 2, 1],
#      [5, 6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose size is equal to the size of `data`'s first dimension. Values should be sorted and can be repeated.
- **`name`**: A name for the operation (optional).

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

tf.math.segment_prod

- Contents
- Aliases:

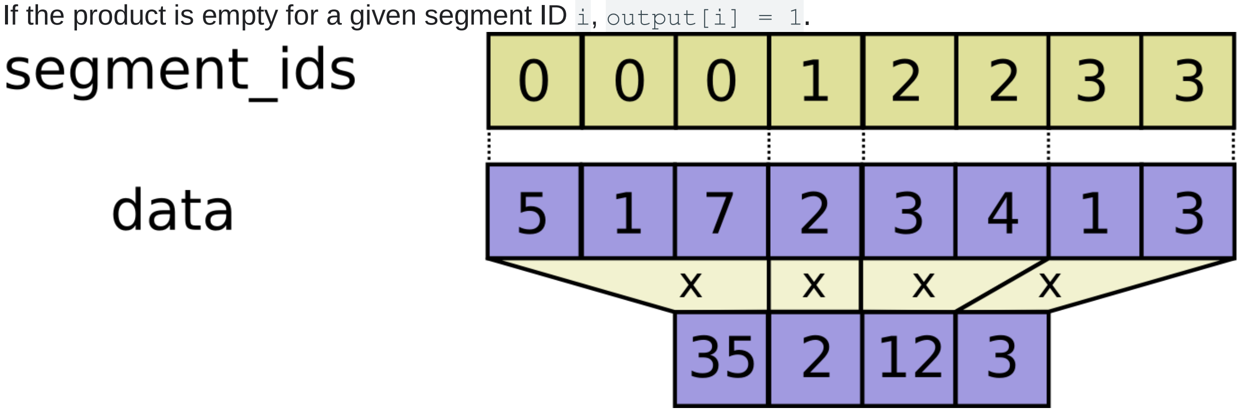
Computes the product along segments of a tensor.

Aliases:

- `tf.compat.v1.math.segment_prod`
- `tf.compat.v1.segment_prod`
- `tf.compat.v2.math.segment_prod`
- `tf.math.segment_prod`

```
tf.math.segment_prod(
    data,
    segment_ids,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
Computes a tensor such that outputi=∏jdataj where the product is over `j` such that `segment_ids[j] == i`.
If the product is empty for a given segment ID `i`, `output[i] = 1`.



For example:

```
c = tf.constant([[1,2,3,4], [4, 3, 2, 1], [5,6,7,8]])
tf.segment_prod(c, tf.constant([0, 0, 1]))
# ==> [[4, 6, 6, 4],
#      [5, 6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose size is equal to the size of `data`'s first dimension. Values should be sorted and can be repeated.
- **`name`**: A name for the operation (optional).

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

tf.math.segment_sum

- Contents
- Aliases:

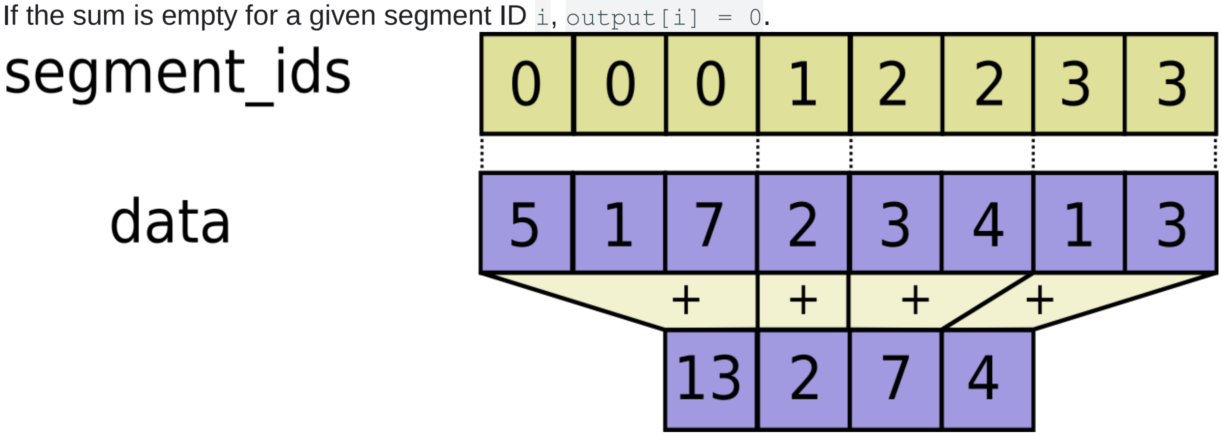
Computes the sum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.segment_sum`
- `tf.compat.v1.segment_sum`
- `tf.compat.v2.math.segment_sum`
- `tf.math.segment_sum`

```
tf.math.segment_sum(
    data,
    segment_ids,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
Computes a tensor such that outputi=∑jdataj where sum is over `j` such that `segment_ids[j] == i`.
If the sum is empty for a given segment ID `i`, `output[i] = 0`.



For example:

```
c = tf.constant([[1,2,3,4], [4, 3, 2, 1], [5,6,7,8]])
tf.segment_sum(c, tf.constant([0, 0, 1]))
# ==> [[5, 5, 5, 5],
#      [5, 6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose size is equal to the size of `data`'s first dimension. Values should be sorted and can be repeated.
- **`name`**: A name for the operation (optional).

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

tf.math.sigmoid

- Contents
- Aliases:
- Used in the tutorials:

Computes sigmoid of `x` element-wise.

Aliases:

- `tf.compat.v1.math.sigmoid`
- `tf.compat.v1.nn.sigmoid`
- `tf.compat.v1.sigmoid`
- `tf.compat.v2.math.sigmoid`
- `tf.compat.v2.nn.sigmoid`
- `tf.compat.v2.sigmoid`
- `tf.math.sigmoid`
- `tf.nn.sigmoid`
- `tf.sigmoid`

```
tf.math.sigmoid(
    x,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Used in the tutorials:

- Convolutional Variational Autoencoder

Specifically, `y = 1 / (1 + exp(-x))`.

Args:

- **`x`**: A Tensor with type `float16`, `float32`, `float64`, `complex64`, or `complex128`.
- **`name`**: A name for the operation (optional).

Returns:
A Tensor with the same type as `x`.

Scipy Compatibility
Equivalent to scipy.special.expit

tf.math.sign

- Contents
- Aliases:

Returns an element-wise indication of the sign of a number.

Aliases:

- `tf.compat.v1.math.sign`
- `tf.compat.v1.sign`
- `tf.compat.v2.math.sign`
- `tf.compat.v2.sign`
- `tf.math.sign`
- `tf.sign`

```
tf.math.sign(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
`y = sign(x) = -1` if `x < 0`; 0 if `x == 0`; 1 if `x > 0`.
For complex numbers, `y = sign(x) = x / |x|` if `x != 0`, otherwise `y = 0`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `x`.
If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.sign(x.values, ...), x.dense_shape)`

tf.math.sin

- Contents
- Aliases:

Computes sin of x element-wise.

Aliases:

- `tf.compat.v1.math.sin`
- `tf.compat.v1.sin`
- `tf.compat.v2.math.sin`
- `tf.compat.v2.sin`
- `tf.math.sin`
- `tf.sin`

```
tf.math.sin(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.sinh

- Contents
- Aliases:

Computes hyperbolic sine of x element-wise.

Aliases:

- `tf.compat.v1.math.sinh`

- `tf.compat.v1.sinh`
- `tf.compat.v2.math.sinh`
- `tf.compat.v2.sinh`
- `tf.math.sinh`
- `tf.sinh`

```
tf.math.sinh(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.softplus

- Contents
- Aliases:

Computes softplus: `log(exp(features) + 1)`.

Aliases:

- `tf.compat.v1.math.softplus`
- `tf.compat.v1.nn.softplus`
- `tf.compat.v2.math.softplus`
- `tf.compat.v2.nn.softplus`
- `tf.math.softplus`
- `tf.nn.softplus`

```
tf.math.softplus(
    features,
    name=None
)
```

Defined in generated file: `python/ops/gen_nn_ops.py`.

Args:

- **`features`**: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.sqrt

- Contents
- Aliases:
- Used in the tutorials:

Computes square root of x element-wise.

Aliases:

- `tf.compat.v1.math.sqrt`
- `tf.compat.v1.sqrt`
- `tf.compat.v2.math.sqrt`
- `tf.compat.v2.sqrt`
- `tf.math.sqrt`
- `tf.sqrt`

```
tf.math.sqrt(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Transformer model for language understanding

I.e., y=x=x1/2.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.sqrt(x.values, ...), x.dense_shape)`

tf.math.square

- Contents
- Aliases:
- Used in the guide:
- Used in the tutorials:

Computes square of x element-wise.

Aliases:

- `tf.compat.v1.math.square`
- `tf.compat.v1.square`
- `tf.compat.v2.math.square`
- `tf.compat.v2.square`
- `tf.math.square`
- `tf.square`

```
tf.math.square(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the guide:

- Eager essentials
- Writing layers and models with TensorFlow Keras

Used in the tutorials:

- [Custom training: basics]
- [Tensors and Operations]

I.e., y=x∗x=x2.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.square(x.values, ...), x.dense_shape)`

tf.math.squared_difference

- Contents
- Aliases:

Returns (x - y)(x - y) element-wise.

Aliases:

- `tf.compat.v1.math.squared_difference`
- `tf.compat.v1.squared_difference`
- `tf.compat.v2.math.squared_difference`
- `tf.math.squared_difference`

```
tf.math.squared_difference(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

NOTE: `math.squared_difference` supports broadcasting. More about broadcasting [here]

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `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`.

tf.math.subtract

- Contents
- Aliases:

Returns x - y element-wise.

Aliases:

- `tf.RaggedTensor.__sub__`
- `tf.compat.v1.RaggedTensor.__sub__`
- `tf.compat.v1.math.subtract`
- `tf.compat.v1.subtract`

- `tf.compat.v2.RaggedTensor.__sub__`
- `tf.compat.v2.math.subtract`
- `tf.compat.v2.subtract`
- `tf.math.subtract`
- `tf.subtract`

```
tf.math.subtract(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.
NOTE: `Subtract` supports broadcasting. More about broadcasting here

Args:
- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `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`.

tf.math.tan

- Contents
- Aliases:

Computes tan of x element-wise.

Aliases:
- `tf.compat.v1.math.tan`
- `tf.compat.v1.tan`
- `tf.compat.v2.math.tan`
- `tf.compat.v2.tan`
- `tf.math.tan`
- `tf.tan`

```
tf.math.tan(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:
- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

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

tf.math.tanh

- Contents

- Aliases:
- Used in the tutorials:

Computes hyperbolic tangent of `x` element-wise.

Aliases:

- `tf.compat.v1.math.tanh`
- `tf.compat.v1.nn.tanh`
- `tf.compat.v1.tanh`
- `tf.compat.v2.math.tanh`
- `tf.compat.v2.nn.tanh`
- `tf.compat.v2.tanh`
- `tf.math.tanh`
- `tf.nn.tanh`
- `tf.tanh`

```
tf.math.tanh(
    x,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Used in the tutorials:

- Image Captioning with Attention
- Neural Machine Translation with Attention
- tf.function

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `bfloat16`, `half`, `float32`, `float64`, `complex64`, `complex128`.
- **`name`**: A name for the operation (optional).

Returns:

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

If `x` is a `SparseTensor`, returns `SparseTensor(x.indices, tf.math.tanh(x.values, ...), x.dense_shape)`

tf.math.top_k

- Contents
- Aliases:

Finds values and indices of the `k` largest entries for the last dimension.

Aliases:

- `tf.compat.v1.math.top_k`
- `tf.compat.v1.nn.top_k`
- `tf.compat.v2.math.top_k`
- `tf.compat.v2.nn.top_k`
- `tf.math.top_k`
- `tf.nn.top_k`

```
tf.math.top_k(
    input,
    k=1,
    sorted=True,
```

```
    name=None
)
```

Defined in `python/ops/nn_ops.py`.

If the input is a vector (rank=1), finds the `k` largest entries in the vector and outputs their values and indices as vectors. Thus `values[j]` is the `j`-th largest entry in `input`, and its index is `indices[j]`.

For matrices (resp. higher rank input), computes the top `k` entries in each row (resp. vector along the last dimension). Thus,

```
values.shape = indices.shape = input.shape[:-1] + [k]
```

If two elements are equal, the lower-index element appears first.

Args:

- **`input`**: 1-D or higher `Tensor` with last dimension at least `k`.
- **`k`**: 0-D `int32` `Tensor`. Number of top elements to look for along the last dimension (along each row for matrices).
- **`sorted`**: If true the resulting `k` elements will be sorted by the values in descending order.
- **`name`**: Optional name for the operation.

Returns:

- **`values`**: The `k` largest elements along each last dimensional slice.
- **`indices`**: The indices of `values` within the last dimension of `input`.

tf.math.truediv

- Contents
- Aliases:

Divides x / y elementwise (using Python 3 division operator semantics).

Aliases:

- `tf.RaggedTensor.__truediv__`
- `tf.compat.v1.RaggedTensor.__truediv__`
- `tf.compat.v1.math.truediv`
- `tf.compat.v1.truediv`
- `tf.compat.v2.RaggedTensor.__truediv__`
- `tf.compat.v2.math.truediv`
- `tf.compat.v2.truediv`
- `tf.math.truediv`
- `tf.truediv`

```
tf.math.truediv(
    x,
    y,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

NOTE: Prefer using the Tensor operator or tf.divide which obey Python division operator semantics.

This function forces Python 3 division operator semantics where all integer arguments are cast to floating types first. This op is generated by normal `x / y` division in Python 3 and in Python 2.7 with`from __future__ import division`. If you want integer division that rounds down, use `x // y` or `tf.math.floordiv`.

`x` and `y` must have the same numeric type. If the inputs are floating point, the output will have the same type. If the inputs are integral, the inputs are cast to `float32` for `int8` and `int16` and `float64` for `int32` and `int64` (matching the behavior of Numpy).

Args:

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

Returns:

`x / y` evaluated in floating point.

Raises:

- **`TypeError`**: If `x` and `y` have different dtypes.

tf.math.unsorted_segment_max

- Contents
- Aliases:

Computes the maximum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.unsorted_segment_max`
- `tf.compat.v1.unsorted_segment_max`
- `tf.compat.v2.math.unsorted_segment_max`
- `tf.math.unsorted_segment_max`

```
tf.math.unsorted_segment_max(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

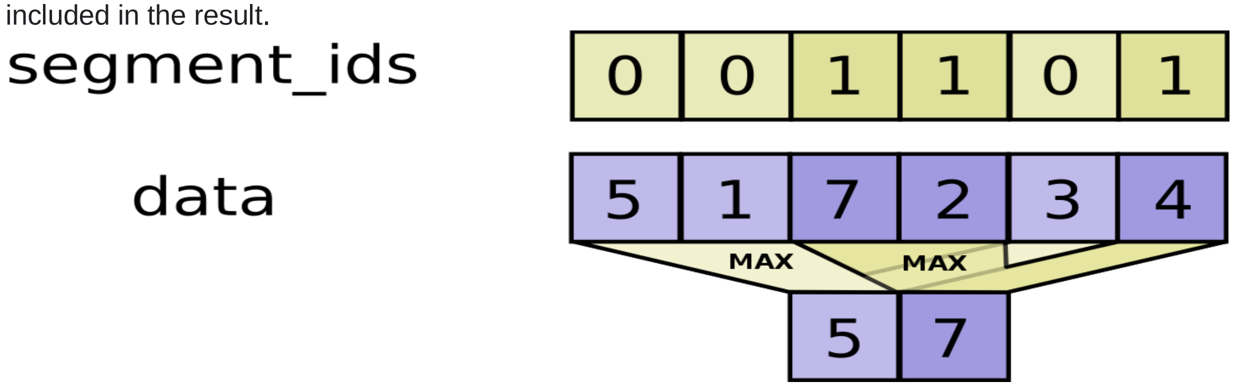
Read the section on segmentation for an explanation of segments.

This operator is similar to the unsorted segment sum operator found (here). Instead of computing the sum over segments, it computes the maximum such that:

outputi=maxj...data[j...] where max is over tuples `j...` such that `segment_ids[j...] == i`.

If the maximum is empty for a given segment ID `i`, it outputs the smallest possible value for the specific numeric type, `output[i] = numeric_limits<T>::lowest()`.

If the given segment ID `i` is negative, then the corresponding value is dropped, and will not be included in the result.



For example:

```
c = tf.constant([[1,2,3,4], [5,6,7,8], [4,3,2,1]])
tf.unsorted_segment_max(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 4,  3, 3, 4],
#       [5,  6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.unsorted_segment_mean

- Contents
- Aliases:

Computes the mean along segments of a tensor.

Aliases:

- `tf.compat.v1.math.unsorted_segment_mean`
- `tf.compat.v1.unsorted_segment_mean`
- `tf.compat.v2.math.unsorted_segment_mean`
- `tf.math.unsorted_segment_mean`

```
tf.math.unsorted_segment_mean(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in `python/ops/math_ops.py`.

Read the section on segmentation for an explanation of segments.

This operator is similar to the unsorted segment sum operator found here. Instead of computing the sum over segments, it computes the mean of all entries belonging to a segment such that: outputi=1/Ni∑j...data[j...] where the sum is over tuples `j...` such that `segment_ids[j...] == i` with \N_i\ being the number of occurrences of id \i\.

If there is no entry for a given segment ID `i`, it outputs 0.

If the given segment ID `i` is negative, the value is dropped and will not be added to the sum of the segment.

Args:

- **`data`**: A `Tensor` with floating point or complex dtype.
- **`segment_ids`**: An integer tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: An integer scalar `Tensor`. The number of distinct segment IDs.
- **`name`**: A name for the operation (optional).

Returns:
A `Tensor`. Has same shape as data, except for the first `segment_ids.rank` dimensions, which are replaced with a single dimension which has size `num_segments`.

tf.math.unsorted_segment_min

- Contents
- Aliases:

Computes the minimum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.unsorted_segment_min`
- `tf.compat.v1.unsorted_segment_min`
- `tf.compat.v2.math.unsorted_segment_min`
- `tf.math.unsorted_segment_min`

```
tf.math.unsorted_segment_min(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
This operator is similar to the unsorted segment sum operator found (here). Instead of computing the sum over segments, it computes the minimum such that:
outputi=minj...data[j...] where min is over tuples `j...` such that `segment_ids[j...] == i`.
If the minimum is empty for a given segment ID `i`, it outputs the largest possible value for the specific numeric type, `output[i] = numeric_limits<T>::max()`.

For example:

```
c = tf.constant([[1,2,3,4], [5,6,7,8], [4,3,2,1]])
tf.unsorted_segment_min(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 1,  2, 2, 1],
#        [5,  6, 7, 8]]
```

If the given segment ID `i` is negative, then the corresponding value is dropped, and will not be included in the result.

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `int64`, `bfloat16`, `uint16`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- **`name`**: A name for the operation (optional).

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

tf.math.unsorted_segment_prod

- Contents

- Aliases:
  Computes the product along segments of a tensor.

  Aliases:

- `tf.compat.v1.math.unsorted_segment_prod`
- `tf.compat.v1.unsorted_segment_prod`
- `tf.compat.v2.math.unsorted_segment_prod`
- `tf.math.unsorted_segment_prod`

```
tf.math.unsorted_segment_prod(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
This operator is similar to the unsorted segment sum operator found (here). Instead of computing the sum over segments, it computes the product of all entries belonging to a segment such that:
outputi=∏j...data[j...] where the product is over tuples `j...` such that `segment_ids[j...] == i`.

For example:

```
c = tf.constant([[1,2,3,4], [5,6,7,8], [4,3,2,1]])
tf.unsorted_segment_prod(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 4,  6, 6, 4],
#       [5,  6, 7, 8]]
```

If there is no entry for a given segment ID `i`, it outputs 1.
If the given segment ID `i` is negative, then the corresponding value is dropped, and will not be included in the result.

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- **`name`**: A name for the operation (optional).

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

tf.math.unsorted_segment_prod

- Contents
- Aliases:
  Computes the product along segments of a tensor.

  Aliases:

- `tf.compat.v1.math.unsorted_segment_prod`
- `tf.compat.v1.unsorted_segment_prod`
- `tf.compat.v2.math.unsorted_segment_prod`
- `tf.math.unsorted_segment_prod`

```
tf.math.unsorted_segment_prod(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read the section on segmentation for an explanation of segments.
This operator is similar to the unsorted segment sum operator found (here). Instead of computing the sum over segments, it computes the product of all entries belonging to a segment such that: $output_i = \prod_{j...} data[j...]$ where the product is over tuples `j...` such that `segment_ids[j...] == i`.

For example:

```
c = tf.constant([[1,2,3,4], [5,6,7,8], [4,3,2,1]])
tf.unsorted_segment_prod(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 4,  6, 6, 4],
#       [5,  6, 7, 8]]
```

If there is no entry for a given segment ID `i`, it outputs 1.
If the given segment ID `i` is negative, then the corresponding value is dropped, and will not be included in the result.

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- **`name`**: A name for the operation (optional).

Returns:

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

tf.math.unsorted_segment_sum

- Contents
- Aliases:

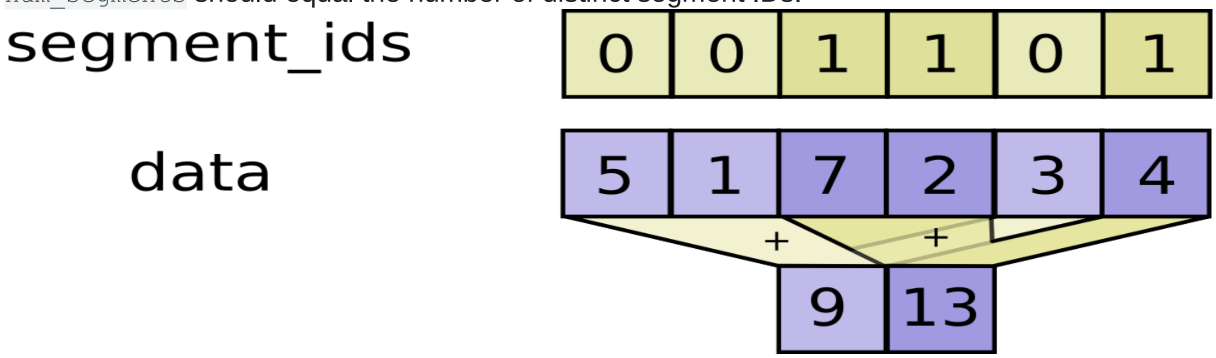
Computes the sum along segments of a tensor.

Aliases:

- `tf.compat.v1.math.unsorted_segment_sum`
- `tf.compat.v1.unsorted_segment_sum`
- `tf.compat.v2.math.unsorted_segment_sum`
- `tf.math.unsorted_segment_sum`

```
tf.math.unsorted_segment_sum(
    data,
    segment_ids,
    num_segments,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
Read [the section on segmentation] for an explanation of segments.
Computes a tensor such that output[i]=∑j...data[j...] where the sum is over tuples `j...` such that `segment_ids[j...] == i`. Unlike `SegmentSum`, `segment_ids` need not be sorted and need not cover all values in the full range of valid values.
If the sum is empty for a given segment ID `i`, `output[i] = 0`. If the given segment ID `i` is negative, the value is dropped and will not be added to the sum of the segment.
`num_segments` should equal the number of distinct segment IDs.




```
c = tf.constant([[1,2,3,4], [5,6,7,8], [4,3,2,1]])
tf.unsorted_segment_sum(c, tf.constant([0, 1, 0]), num_segments=2)
# ==> [[ 5,  5, 5, 5],
#       [5,  6, 7, 8]]
```

Args:

- **`data`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, `int64`, `qint8`, `quint8`, `qint32`, `bfloat16`, `uint16`, `complex128`, `half`, `uint32`, `uint64`.
- **`segment_ids`**: A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor whose shape is a prefix of `data.shape`.
- **`num_segments`**: A `Tensor`. Must be one of the following types: `int32`, `int64`.
- **`name`**: A name for the operation (optional).

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

tf.math.xdivy

- Contents
- Aliases:

Returns 0 if x == 0, and x / y otherwise, elementwise.

Aliases:

- `tf.compat.v1.math.xdivy`
- `tf.compat.v2.math.xdivy`
- `tf.math.xdivy`

```
tf.math.xdivy(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `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`.

tf.math.xlogy

- Contents
- Aliases:

Returns 0 if x == 0, and x * log(y) otherwise, elementwise.

Aliases:

- `tf.compat.v1.math.xlogy`
- `tf.compat.v2.math.xlogy`
- `tf.math.xlogy`

```
tf.math.xlogy(
    x,
    y,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `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`.

tf.math.zero_fraction

- Contents
- Aliases:

Returns the fraction of zeros in `value`.

Aliases:

- `tf.compat.v1.math.zero_fraction`
- `tf.compat.v1.nn.zero_fraction`
- `tf.compat.v2.math.zero_fraction`
- `tf.compat.v2.nn.zero_fraction`
- `tf.math.zero_fraction`
- `tf.nn.zero_fraction`

```
tf.math.zero_fraction(
    value,
    name=None
)
```

Defined in `python/ops/nn_impl.py`.

If `value` is empty, the result is `nan`.
This is useful in summaries to measure and report sparsity. For example,

```
    z = tf.nn.relu(...)
    summ = tf.compat.v1.summary.scalar('sparsity', tf.nn.zero_fraction(z))
```

Args:

- **`value`**: A tensor of numeric type.
- **`name`**: A name for the operation (optional).

Returns:
The fraction of zeros in `value`, with type `float32`.

tf.math.zeta

- Contents
- Aliases:

Compute the Hurwitz zeta function ζ(x,q).

Aliases:

- `tf.compat.v1.math.zeta`
- `tf.compat.v1.zeta`
- `tf.compat.v2.math.zeta`
- `tf.math.zeta`

```
tf.math.zeta(
    x,
    q,
    name=None
)
```

Defined in generated file: `python/ops/gen_math_ops.py`.
The Hurwitz zeta function is defined as:
$\zeta(x,q)=\sum_{n=0}^{\infty}(q+n)^{-x}$

Args:

- **`x`**: A `Tensor`. Must be one of the following types: `float32`, `float64`.
- **`q`**: 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`.