Neural Network

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

Contents

  - `tf.nn.conv1d(value, filters, stride, padding, use_cudnn_on_gpu=None, data_format=None, name=None)`
  - `tf.nn.conv3d(input, filter, strides, padding, name=None)`
  - `tf.nn.conv3d_transpose(value, filter, output_shape, strides, padding=SAME, name=None)`
- Pooling
  - `tf.nn.avg_pool(value, ksize, strides, padding, data_format=NHWC, name=None)`
  - `tf.nn.max_pool(value, ksize, strides, padding, data_format=NHWC, name=None)`
  - `tf.nn.max_pool_with_argmax(input, ksize, strides, padding, Targmax=None, name=None)`
  - `tf.nn.avg_pool3d(input, ksize, strides, padding, name=None)`
  - `tf.nn.max_pool3d(input, ksize, strides, padding, name=None)`
  - `tf.nn.fractional_avg_pool(value, pooling_ratio, pseudo_random=None, overlapping=None, deterministic=None, seed=None, seed2=None, name=None)`
  - `tf.nn.fractional_max_pool(value, pooling_ratio, pseudo_random=None, overlapping=None, deterministic=None, seed=None, seed2=None, name=None)`
- Morphological filtering
  - `tf.nn.dilation2d(input, filter, strides, rates, padding, name=None)`
  - `tf.nn.erosion2d(value, kernel, strides, rates, padding, name=None)`
- Normalization
  - `tf.nn.l2_normalize(x, dim, epsilon=1e-12, name=None)`
  - `tf.nn.local_response_normalization(input, depth_radius=None, bias=None, alpha=None, beta=None, name=None)`
  - `tf.nn.sufficient_statistics(x, axes, shift=None, keep_dims=False, name=None)`
  - `tf.nn.normalize_moments(counts, mean_ss, variance_ss, shift, name=None)`
  - `tf.nn.moments(x, axes, shift=None, name=None, keep_dims=False)`
- Losses
  - `tf.nn.l2_loss(t, name=None)`
  - `tf.nn.log_poisson_loss(log_input, targets, compute_full_loss=False, name=None)`
- Classification

    - `tf.nn.sigmoid_cross_entropy_with_logits(logits, targets, name=None)`
    - `tf.nn.softmax(logits, dim=-1, name=None)`
    - `tf.nn.log_softmax(logits, dim=-1, name=None)`
    - `tf.nn.softmax_cross_entropy_with_logits(logits, labels, dim=-1, name=None)`
    - `tf.nn.sparse_softmax_cross_entropy_with_logits(logits, labels, name=None)`
    - `tf.nn.weighted_cross_entropy_with_logits(logits, targets, pos_weight, name=None)`
- Embeddings
    - `tf.nn.embedding_lookup(params, ids, partition_strategy=mod, name=None, validate_indices=True)`
    - `tf.nn.embedding_lookup_sparse(params, sp_ids, sp_weights, partition_strategy=mod, name=None, combiner=None)`
- Recurrent Neural Networks
    - `tf.nn.dynamic_rnn(cell, inputs, sequence_length=None, initial_state=None, dtype=None, parallel_iterations=None, swap_memory=False, time_major=False, scope=None)`
    - `tf.nn.rnn(cell, inputs, initial_state=None, dtype=None, sequence_length=None, scope=None)`
    - `tf.nn.state_saving_rnn(cell, inputs, state_saver, state_name, sequence_length=None, scope=None)`
    - `tf.nn.bidirectional_dynamic_rnn(cell_fw, cell_bw, inputs, sequence_length=None, initial_state_fw=None, initial_state_bw=None, dtype=None, parallel_iterations=None, swap_memory=False, time_major=False, scope=None)`
    - `tf.nn.bidirectional_rnn(cell_fw, cell_bw, inputs, initial_state_fw=None, initial_state_bw=None, dtype=None, sequence_length=None, scope=None)`
    - `tf.nn.raw_rnn(cell, loop_fn, parallel_iterations=None, swap_memory=False, scope=None)`
- Conectionist Temporal Classification (CTC)
    - `tf.nn.ctc_loss(inputs, labels, sequence_length, preprocess_collapse_repeated=False, ctc_merge_repeated=True, time_major=True)`
    - `tf.nn.ctc_greedy_decoder(inputs, sequence_length, merge_repeated=True)`
    - `tf.nn.ctc_beam_search_decoder(inputs, sequence_length, beam_width=100, top_paths=1, merge_repeated=True)`

- Evaluation
  - `tf.nn.top_k(input, k=1, sorted=True, name=None)`
  - `tf.nn.in_top_k(predictions, targets, k, name=None)`
- Candidate Sampling
  - Sampled Loss Functions
  - `tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=False, partition_strategy=mod, name=nce_loss)`
  - `tf.nn.sampled_softmax_loss(weights, biases, inputs, labels, num_sampled, num_classes, num_true=1, sampled_values=None, remove_accidental_hits=True, partition_strategy=mod, name=sampled_softmax_loss)`
  - Candidate Samplers
  - `tf.nn.uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
  - `tf.nn.log_uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
  - `tf.nn.learned_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`
  - `tf.nn.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)`
  - Miscellaneous candidate sampling utilities
  - `tf.nn.compute_accidental_hits(true_classes, sampled_candidates, num_true, seed=None, name=None)`
- Other Functions and Classes
  - `tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)`
  - `tf.nn.depthwise_conv2d_native(input, filter, strides, padding, name=None)`

Activation Functions.

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities (`sigmoid`, `tanh`, `elu`, `softplus`, and `softsign`), continuous but not everywhere differentiable functions (`relu`, `relu6`, `crelu` and `relu_x`), and random regularization (`dropout`).

All activation ops apply componentwise, and produce a tensor of the same shape as the input tensor.

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

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

Args:

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

Returns:

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

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

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

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.crelu(features, name=None)`

Computes Concatenated ReLU.

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: https://arxiv.org/abs/1603.05201

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.elu(features, name=None)`

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

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

Args:

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

Returns:

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

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

Computes softplus: `log(exp(features) + 1)`.

Args:

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

Returns:

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

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

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

Args:

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

Returns:

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

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

Computes dropout.

With probability `keep_prob`, outputs the input element scaled up by `1 / keep_prob`, otherwise outputs `0`. The scaling is so that the expected sum is unchanged.

By default, each element is kept or dropped independently. If `noise_shape` is specified, it must be broadcastable 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 tensor.
- `keep_prob`: A scalar `Tensor` with the same type as x. The probability that each element is kept.
- `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 `set_random_seed` for behavior.
- `name`: A name for this operation (optional).

Returns:

A Tensor of the same shape of `x`.

Raises:

- `ValueError`: If `keep_prob` is not in `(0, 1]`.

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

Adds `bias` to `value`.

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. 'NHWC' and 'NCHW' are supported.
- `name`: A name for the operation (optional).

Returns:

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

`tf.sigmoid(x, name=None)`

Computes sigmoid of `x` element-wise.

Specifically, `y = 1 / (1 + exp(-x))`.

Args:

- `x`: A Tensor with type `float32`, `float64`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

`tf.tanh(x, name=None)`

Computes hyperbolic tangent of `x` element-wise.

Args:

- `x`: A Tensor or SparseTensor with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Returns:

A Tensor or SparseTensor respectively with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

Convolution

The convolution ops sweep a 2-D filter over a batch of images, applying the filter to each window of each image of the appropriate size. The different ops trade off between generic vs. specific filters:

- `conv2d`: Arbitrary filters that can mix channels together.
- `depthwise_conv2d`: Filters that operate on each channel independently.
- `separable_conv2d`: A depthwise spatial filter followed by a pointwise filter.

Note that although these ops are called "convolution", they are strictly speaking "cross-correlation" since the filter is combined with an input window without reversing the filter. For details, see the properties of cross-correlation.

The filter is applied to image patches of the same size as the filter and strided according to the `strides` argument. `strides = [1, 1, 1, 1]` applies the filter to a patch at every offset, `strides = [1, 2, 2, 1]` applies the filter to every other image patch in each dimension, etc.

Ignoring channels for the moment, and assume that the 4-D `input` has shape `[batch, in_height, in_width, ...]` and the 4-D `filter` has shape `[filter_height, filter_width, ...]`, then the spatial semantics of the convolution ops are as follows: first, according to the padding scheme chosen as `'SAME'` or `'VALID'`, the output size and the padding pixels are computed. For the `'SAME'` padding, the output height and width are computed as:

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width  = ceil(float(in_width) / float(strides[2]))
```

and the padding on the top and left are computed as:

```
pad_along_height = ((out_height - 1) * strides[1] +
                    filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                   filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2
```

Note that the division by 2 means that there might be cases when the padding on both sides (top vs bottom, right vs left) are off by one. In this case, the bottom and right sides always get the one additional padded pixel. For example, when `pad_along_height` is 5, we pad 2 pixels at the top and 3 pixels at the bottom. Note that this is different from existing libraries such as cuDNN and Caffe, which explicitly specify the number of padded pixels and always pad the same number of pixels on both sides.

For the `'VALID`' padding, the output height and width are computed as:

```
out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))
```

and the padding values are always zero. The output is then computed as

```
output[b, i, j, :] =
    sum_{di, dj} input[b, strides[1] * i + di - pad_top,
                       strides[2] * j + dj - pad_left, ...] *
                 filter[di, dj, ...]
```

where any value outside the original input image region are considered zero (i.e. we pad zero values around the border of the image).

Since `input` is 4-D, each `input[b, i, j, :]` is a vector. For `conv2d`, these vectors are multiplied by the `filter[di, dj, :, :]` matrices to produce new vectors. For `depthwise_conv_2d`, each scalar component `input[b, i, j, k]` is multiplied by a vector `filter[di, dj, k]`, and all the vectors are concatenated.

`tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)`

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

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`, `float32`, `float64`.
- `filter`: A `Tensor`. Must have the same type as `input`.
- `strides`: A list of `ints`. 1-D of length 4. The stride of the sliding window for each dimension of `input`. Must be in the same order as the dimension specified with format.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `use_cudnn_on_gpu`: An optional `bool`. Defaults to `True`.
- `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, in_height, in_width, in_channels]. Alternatively, the format could be "NCHW", the data storage order of: [batch, in_channels, in_height, in_width].
- `name`: A name for the operation (optional).

Returns:

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

`tf.nn.depthwise_conv2d(input, filter, strides, padding, name=None)`

Depthwise 2-D convolution.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` 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,

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

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

Args:

- `input`: 4-D with shape `[batch, in_height, in_width, in_channels]`.
- `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 comment here
- `name`: A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape `[batch, out_height, out_width, in_channels * channel_multiplier]`.

`tf.nn.separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding, name=None)`

2-D convolution with separable filters.

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,

```
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]`.

Args:

- `input`: 4-D `Tensor` with shape `[batch, in_height, in_width, in_channels]`.
- `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 comment here
- `name`: A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape `[batch, out_height, out_width, out_channels]`.

Raises:

- `ValueError`: If channel_multiplier * in_channels > out_channels, which means that the separable convolution is overparameterized.

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

Atrous convolution (a.k.a. convolution with holes or dilated 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[b, i, j, k] = sum_{di, dj, q} filters[di, dj, q, k] *
      value[b, i + rate * di, j + rate * dj, q]
```

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`.

Raises:

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

`tf.nn.conv2d_transpose(value, filter, output_shape, strides, padding='SAME', name=None)`

The transpose of `conv2d`.

This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is actually the transpose (gradient) of `conv2d` rather than an actual deconvolution.

Args:

- `value`: A 4-D `Tensor` of type `float` and shape `[batch, height, width, in_channels]`.

- `filter`: A 4-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`: A list of ints. 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 comment here
- `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, or if padding is other than `'VALID'` or `'SAME'`.

`tf.nn.conv1d(value, filters, stride, padding, use_cudnn_on_gpu=None, data_format=None, name=None)`

Computes a 1-D convolution given 3-D input and filter tensors.

Given an input tensor of shape [batch, in_width, in_channels] and a filter / kernel tensor of shape [filter_width, in_channels, out_channels], this op reshapes the arguments to pass them to conv2d to perform the equivalent convolution operation.

Internally, this op reshapes the input tensors and invokes `tf.nn.conv2d`. A tensor of shape [batch, in_width, in_channels] is reshaped to [batch, 1, in_width, in_channels], and the filter is reshaped to [1, filter_width, in_channels, out_channels]. The result is then reshaped back to batch, out_width, out_channels and returned to the caller.

Args:

- `value`: A 3D `Tensor`. Must be of type `float32` or `float64`.
- `filters`: A 3D `Tensor`. Must have the same type as `input`.

- `stride`: An `integer`. The number of entries by which the filter is moved right at each step.
- `padding`: 'SAME' or 'VALID'
- `use_cudnn_on_gpu`: An optional `bool`. Defaults to `True`.
- `data_format`: An optional `string` from `"NHWC", "NCHW"`. Defaults to `"NHWC"`, the data is stored in the order of [batch, in_width, in_channels]. The `"NCHW"` format stores data as [batch, in_channels, in_width].
- `name`: A name for the operation (optional).

Returns:

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

`tf.nn.conv3d(input, filter, strides, padding, name=None)`

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

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: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Shape `[batch, in_depth, in_height, in_width, in_channels]`.
- `filter`: 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 `filter`.
- `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.
- `name`: A name for the operation (optional).

Returns:

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

`tf.nn.conv3d_transpose(value, filter, output_shape, strides, padding='SAME', name=None)`

The transpose of `conv3d`.

This operation is sometimes called "deconvolution" after Deconvolutional Networks, but is actually the transpose (gradient) of `conv3d` rather than an actual deconvolution.

Args:

- `value`: A 5-D `Tensor` of type `float` and shape `[batch, depth, height, width, in_channels]`.
- `filter`: A 5-D `Tensor` with the same type as `value` and shape `[depth, 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`: A list of ints. 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 comment here
- `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, or if padding is other than `'VALID'` or `'SAME'`.

Pooling

The pooling ops sweep a rectangular window over the input tensor, computing a reduction operation for each window (average, max, or max with argmax). Each pooling op uses rectangular windows of size `ksize` separated by offset `strides`. For example, if `strides` is all ones every window is used, if `strides` is all twos every other window is used in each dimension, etc.

In detail, the output is

```
output[i] = reduce(value[strides * i:strides * i + ksize])
```

where the indices also take into consideration the padding values. Please refer to the `Convolution` section for details about the padding calculation.

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

Performs the average pooling on the input.

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

Args:

- `value`: A 4-D `Tensor` of shape `[batch, height, width, channels]` and type `float32`, `float64`, `qint8`, `quint8`, or `qint32`.
- `ksize`: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.
- `strides`: A list of ints that has length >= 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 comment here
- `data_format`: A string. 'NHWC' and 'NCHW' are supported.
- `name`: Optional name for the operation.

Returns:

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

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

Performs the max pooling on the input.

Args:

- `value`: A 4-D `Tensor` with shape `[batch, height, width, channels]` and type `tf.float32`.
- `ksize`: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.
- `strides`: A list of ints that has length >= 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 comment here
- `data_format`: A string. 'NHWC' and 'NCHW' are supported.
- `name`: Optional name for the operation.

Returns:

A `Tensor` with type `tf.float32`. The max pooled output tensor.

`tf.nn.max_pool_with_argmax(input, ksize, strides, padding, Targmax=None, name=None)`

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

The indices in `argmax` are flattened, so that a maximum value at position `[b, y, x, c]` becomes flattened index `((b * height + y) * width + x) * channels + c`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `half`. 4-D with shape `[batch, height, width, channels]`. Input to pool over.
- `ksize`: A list of `ints` that has length `>= 4`. The size of the window for each dimension of the input tensor.
- `strides`: A list of `ints` that has length `>= 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.
- `Targmax`: An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`.
- `name`: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (output, argmax).

- `output`: A `Tensor`. Has the same type as `input`. The max pooled output tensor.
- `argmax`: A `Tensor` of type `Targmax`. 4-D. The flattened indices of the max values chosen for each output.

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

Performs 3D average pooling on the input.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Shape `[batch, depth, rows, cols, channels]` tensor to pool over.
- `ksize`: A list of `ints` that has length `>= 5`. 1-D tensor of length 5. The size of the window for each dimension of the input tensor. Must have `ksize[0] = ksize[4] = 1`.
- `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.

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

Returns:

A `Tensor`. Has the same type as `input`. The average pooled output tensor.

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

Performs 3D max pooling on the input.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Shape `[batch, depth, rows, cols, channels]` tensor to pool over.
- `ksize`: A list of `ints` that has length `>= 5`. 1-D tensor of length 5. The size of the window for each dimension of the input tensor. Must have `ksize[0] = ksize[4] = 1`.
- `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.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. The max pooled output tensor.

`tf.nn.fractional_avg_pool(value, pooling_ratio, pseudo_random=None, overlapping=None, deterministic=None, seed=None, seed2=None, name=None)`

Performs fractional average pooling on the input.

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`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`. 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 [41/3, 26/3] for fractional avg pooling.

  `deterministic`: An optional `bool`. Defaults to `False`. When set to True, a fixed pooling region will be used when iterating over a FractionalAvgPool node in the computation graph. Mainly used in unit test to make FractionalAvgPool deterministic.

  `seed`: An optional `int`. Defaults to `0`. If either seed or seed2 are set to be non-zero, the random number generator is seeded by the given seed. Otherwise, it is seeded by a random seed.

  `seed2`: An optional `int`. Defaults to `0`. An second seed to avoid seed collision.

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

Returns:

A tuple of `Tensor` objects (output, row_pooling_sequence, col_pooling_sequence).

- `output`: A `Tensor`. Has the same type as `value`. output tensor after fractional avg pooling.
- `row_pooling_sequence`: A `Tensor` of type `int64`. row pooling sequence, needed to calculate gradient.
- `col_pooling_sequence`: A `Tensor` of type `int64`. column pooling sequence, needed to calculate gradient.

`tf.nn.fractional_max_pool(value, pooling_ratio, pseudo_random=None, overlapping=None, deterministic=None, seed=None, seed2=None, name=None)`

Performs fractional max pooling on the input.

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`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`. 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 max pooling.

  `deterministic`: An optional `bool`. Defaults to `False`. When set to True, a fixed pooling region will be used when iterating over a FractionalMaxPool node in the computation graph. Mainly used in unit test to make FractionalMaxPool deterministic.

  `seed`: An optional `int`. Defaults to `0`. If either seed or seed2 are set to be non-zero, the random number generator is seeded by the given seed. Otherwise, it is seeded by a random seed.

  `seed2`: An optional `int`. Defaults to `0`. An second seed to avoid seed collision.

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

Returns:

A tuple of `Tensor` objects (output, row_pooling_sequence, col_pooling_sequence).

- `output`: A `Tensor`. Has the same type as `value`. output tensor after fractional max pooling.

- `row_pooling_sequence`: A `Tensor` of type `int64`. row pooling sequence, needed to calculate gradient.
- `col_pooling_sequence`: A `Tensor` of type `int64`. column pooling sequence, needed to calculate gradient.

Morphological filtering

Morphological operators are non-linear filters used in image processing.

Greyscale morphological dilation is the max-sum counterpart of standard sum-product convolution:

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

The `filter` is usually called structuring function. Max-pooling is a special case of greyscale morphological dilation when the filter assumes all-zero values (a.k.a. flat structuring function).

Greyscale morphological erosion is the min-sum counterpart of standard sum-product convolution:

```
output[b, y, x, c] =
    min_{dy, dx} input[b,
                       strides[1] * y - rates[1] * dy,
                       strides[2] * x - rates[2] * dx,
                       c] -
                 filter[dy, dx, c]
```

Dilation and erosion are dual to each other. The dilation of the input signal `f` by the structuring signal `g` is equal to the negation of the erosion of `-f` by the reflected `g`, and vice versa.

Striding and padding is carried out in exactly the same way as in standard convolution. Please refer to the `Convolution` section for details.

`tf.nn.dilation2d(input, filter, strides, rates, padding, name=None)`

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

The `input` tensor has shape `[batch, in_height, in_width, depth]` and the `filter` 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] +
                filter[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 `filter` is equal to the negation of the erosion of `-input` by the reflected `filter`.

Args:

- `input`: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`. 4-D with shape `[batch, in_height, in_width, depth]`.
- `filter`: 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]`.
- `rates`: A list of `ints` that has length `>= 4`. The input stride for atrous morphological dilation. Must be: `[1, rate_height, rate_width, 1]`.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.

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

Returns:

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

`tf.nn.erosion2d(value, kernel, strides, rates, padding, name=None)`

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

The `value` tensor has shape `[batch, in_height, in_width, depth]` and the `kernel` tensor has shape `[kernel_height, kernel_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 - rates[1] * dy,
                      strides[2] * x - rates[2] * dx,
                      c] -
                kernel[dy, dx, c]
```

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

Args:

- `value`: A `Tensor`. 4-D with shape `[batch, in_height, in_width, depth]`.
- `kernel`: A `Tensor`. Must have the same type as `value`. 3-D with shape `[kernel_height, kernel_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]`.

- `rates`: 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]`.
- `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `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 `kernel`' shape, or if padding is other than `'VALID'` or `'SAME'`.

Normalization

Normalization is useful to prevent neurons from saturating when inputs may have varying scale, and to aid generalization.

`tf.nn.l2_normalize(x, dim, epsilon=1e-12, name=None)`

Normalizes along dimension `dim` using an L2 norm.

For a 1-D tensor with `dim = 0`, computes

```
output = x / sqrt(max(sum(x**2), epsilon))
```

For `x` with more dimensions, independently normalizes each 1-D slice along dimension `dim`.

Args:

- `x`: A `Tensor`.
- `dim`: 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.nn.local_response_normalization(input, depth_radius=None, bias=None, alpha=None, beta=None, name=None)`

Local Response Normalization.

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: `float32`, `half`. 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.sufficient_statistics(x, axes, shift=None, keep_dims=False, name=None)`

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

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.
- `keep_dims`: 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.normalize_moments(counts, mean_ss, variance_ss, shift, name=None)`

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

Args:

- `counts`: A `Tensor` containing a 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.moments(x, axes, shift=None, name=None, keep_dims=False)`

Calculate the mean and variance of `x`.

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.

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`: 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.
- `name`: Name used to scope the operations that compute the moments.
- `keep_dims`: produce moments with the same dimensionality as the input.

Returns:

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

Losses

The loss ops measure error between two tensors, or between a tensor and zero. These can be used for measuring accuracy of a network in a regression task or for regularization purposes (weight decay).

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

L2 Loss.

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: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Typically 2-D, but may have any dimensions.
- `name`: A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `t`. 0-D.

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

Computes log poisson loss given `log_input`.

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:

- `log_input`: A `Tensor` of type `float32` or `float64`.
- `targets`: A `Tensor` of the same type and shape as `log_input`.
- `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.

Classification

TensorFlow provides several operations that help you perform classification.

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

Computes sigmoid cross entropy given `logits`.

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 = targets`. 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 `targets` must have the same type and shape.

Args:

- `logits`: A `Tensor` of type `float32` or `float64`.
- `targets`: A `Tensor` of the same type and shape as `logits`.
- `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 `targets` do not have the same shape.

`tf.nn.softmax(logits, dim=-1, name=None)`

Computes log softmax activations.

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

```
softmax = exp(logits) / reduce_sum(exp(logits), dim)
```

Args:

- `logits`: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `dim`: 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 `dim` is beyond the last dimension of `logits`.

`tf.nn.log_softmax(logits, dim=-1, name=None)`

Computes log softmax activations.

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

```
logsoftmax = logits - reduce_sum(exp(logits), dim)
```

Args:

- `logits`: A non-empty `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `dim`: 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 `dim` is beyond the last dimension of `logits`.

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

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

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.

`logits` and `labels` must have the same shape `[batch_size, num_classes]` and the same dtype (either `float16`, `float32`, or `float64`).

Args:

- `logits`: Unscaled log probabilities.
- `labels`: Each row `labels[i]` must be a valid probability distribution.
- `dim`: The class dimension. Defaulted to -1 which is the last dimension.
- `name`: A name for the operation (optional).

Returns:

A 1-D `Tensor` of length `batch_size` of the same type as `logits` with the softmax cross entropy loss.

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

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

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`.

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 labels of shape `[batch_size]`. But higher dimensions are supported.

Args:

logits: Unscaled log probabilities of rank `r` and shape `[d_0, d_1, ..., d_{r-2}, num_classes]` and dtype `float32` or `float64`. labels: `Tensor` of shape `[d_0, d_1, ..., d_{r-2}]` 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 corresponding loss and gradient rows on GPU. 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 labels minus one.

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

Computes a weighted cross entropy.

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:

targets * -log(sigmoid(logits)) + (1 - targets) * -log(1 - sigmoid(logits))

The argument `pos_weight` is used as a multiplier for the positive targets:

targets * -log(sigmoid(logits)) * pos_weight + (1 - targets) * -log(1 - sigmoid(logits))

For brevity, let `x = logits`, `z = targets`, `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 `targets` must have the same type and shape.

Args:

- `logits`: A `Tensor` of type `float32` or `float64`.
- `targets`: A `Tensor` of the same type and shape as `logits`.
- `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 weightedlogistic losses.

Raises:

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

Embeddings

TensorFlow provides library support for looking up values in embedding tensors.

`tf.nn.embedding_lookup(params, ids, partition_strategy='mod', name=None, validate_indices=True)`

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

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 partition of a larger embedding tensor.

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.

If `partition_strategy` is `"mod"`, we assign each id to partition `p = id % len(params)`. For instance, 13 ids are split across 5 partitions as: `[[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8], [4, 9]]`

If `partition_strategy` is `"div"`, we assign ids to partitions in a contiguous manner. In this case, 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 list of tensors with the same type and which can be concatenated along dimension 0. Each `Tensor` must be appropriately sized for the given `partition_strategy`.
- `ids`: A `Tensor` with type `int32` or `int64` containing the ids to be looked up in `params`.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(params) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`.
- `name`: A name for the operation (optional).
- `validate_indices`: Whether or not to validate gather indices.

Returns:

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

Raises:

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

`tf.nn.embedding_lookup_sparse(params, sp_ids, sp_weights, partition_strategy='mod', name=None, combiner=None)`

Computes embeddings for the given ids and weights.

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.
- `sp_ids`: N x M SparseTensor of int64 ids (typically from FeatureValueToId), 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.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(params) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name`: Optional name for the op.
- `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.

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 output[2, :] = params[1, :] * 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"}.

Recurrent Neural Networks

TensorFlow provides a number of methods for constructing Recurrent Neural Networks. Most accept an `RNNCell`-subclassed object (see the documentation for `tf.nn.rnn_cell`).

`tf.nn.dynamic_rnn(cell, inputs, sequence_length=None, initial_state=None, dtype=None, parallel_iterations=None, swap_memory=False, time_major=False, scope=None)`

Creates a recurrent neural network specified by RNNCell `cell`.

This function is functionally identical to the function `rnn` above, but performs fully dynamic unrolling of `inputs`.

Unlike `rnn`, the input `inputs` is not a Python list of `Tensors`, one for each frame. Instead, `inputs` may be a single `Tensor` where the maximum time is either the first or second dimension (see the parameter `time_major`). Alternatively, it may be a (possibly nested) tuple of Tensors, each of them having matching batch and time dimensions. The corresponding output is either a single `Tensor` having the same number of time steps and batch size, or a (possibly nested) tuple of such tensors, matching the nested structure of `cell.output_size`.

The parameter `sequence_length` is optional and is used to copy-through state and zero-out outputs when past a batch element's sequence length. So it's more for correctness than performance, unlike in rnn().

Args:

- `cell`: An instance of RNNCell.

  `inputs`: The RNN inputs.

  If `time_major == False` (default), this must be a `Tensor` of shape: `[batch_size, max_time, ...]`, or a nested tuple of such elements.

  If `time_major == True`, this must be a `Tensor` of shape: `[max_time, batch_size, ...]`, or a nested tuple of such elements.

  This may also be a (possibly nested) tuple of Tensors satisfying this property. The first two dimensions must match across all the inputs, but otherwise the ranks and other shape components may differ. In this case, input to `cell` at each time-step will replicate the structure of these tuples, except for the time dimension (from which the time is taken).

  The input to `cell` at each time step will be a `Tensor` or (possibly nested) tuple of Tensors each with dimensions `[batch_size, ...]`.

  `sequence_length`: (optional) An int32/int64 vector sized `[batch_size]`.

  `initial_state`: (optional) An initial state for the RNN. If `cell.state_size` is an integer, this must be a `Tensor` of appropriate type and shape `[batch_size, cell.state_size]`. If `cell.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s] for s in cell.state_size`.

`dtype`: (optional) The data type for the initial state and expected output. Required if initial_state is not provided or RNN state has a heterogeneous dtype.

`parallel_iterations`: (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer.

`swap_memory`: Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.

`time_major`: The shape format of the `inputs` and `outputs` Tensors. If true, these `Tensors` must be shaped `[max_time, batch_size, depth]`. If false, these `Tensors` must be shaped `[batch_size, max_time, depth]`. Using `time_major = True` is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.

`scope`: VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (outputs, state) where:

`outputs`: The RNN output `Tensor`.

If time_major == False (default), this will be a `Tensor` shaped: `[batch_size, max_time, cell.output_size]`.

If time_major == True, this will be a `Tensor` shaped: `[max_time, batch_size, cell.output_size]`.

Note, if `cell.output_size` is a (possibly nested) tuple of integers or `TensorShape` objects, then `outputs` will be a tuple having the same structure as `cell.output_size`, containing Tensors having shapes corresponding to the shape data in `cell.output_size`.

`state`: The final state. If `cell.state_size` is an int, this will be shaped `[batch_size, cell.state_size]`. If it is a `TensorShape`, this will be shaped `[batch_size] + cell.state_size`. If it is a (possibly nested) tuple of ints or `TensorShape`, this will be a tuple having the corresponding shapes.

Raises:

- `TypeError`: If `cell` is not an instance of RNNCell.
- `ValueError`: If inputs is None or an empty list.

`tf.nn.rnn(cell, inputs, initial_state=None, dtype=None, sequence_length=None, scope=None)`

Creates a recurrent neural network specified by RNNCell `cell`.

The simplest form of RNN network generated is: `python state = cell.zero_state(...) outputs = [] for input_ in inputs: output, state = cell(input_, state) outputs.append(output) return (outputs, state)` However, a few other options are available:

An initial state can be provided. If the sequence_length vector is provided, dynamic calculation is performed. This method of calculation does not compute the RNN steps past the maximum sequence length of the minibatch (thus saving computational time), and properly propagates the state at an example's sequence length to the final state output.

The dynamic calculation performed is, at time `t` for batch row `b`, `python (output, state)(b, t) = (t >= sequence_length(b)) ? (zeros(cell.output_size), states(b, sequence_length(b) - 1)) : cell(input(b, t), state(b, t - 1))`

Args:

- `cell`: An instance of RNNCell.
- `inputs`: A length T list of inputs, each a `Tensor` of shape `[batch_size, input_size]`, or a nested tuple of such elements.
- `initial_state`: (optional) An initial state for the RNN. If `cell.state_size` is an integer, this must be a `Tensor` of appropriate type and shape `[batch_size, cell.state_size]`. If `cell.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s] for s in cell.state_size`.
- `dtype`: (optional) The data type for the initial state and expected output. Required if initial_state is not provided or RNN state has a heterogeneous dtype.
- `sequence_length`: Specifies the length of each sequence in inputs. An int32 or int64 vector (tensor) size `[batch_size]`, values in `[0, T)`.

- `scope`: VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (outputs, state) where: - outputs is a length T list of outputs (one for each input), or a nested tuple of such elements. - state is the final state

Raises:

- `TypeError`: If `cell` is not an instance of RNNCell.
- `ValueError`: If `inputs` is `None` or an empty list, or if the input depth (column size) cannot be inferred from inputs via shape inference.

`tf.nn.state_saving_rnn(cell, inputs, state_saver, state_name, sequence_length=None, scope=None)`

RNN that accepts a state saver for time-truncated RNN calculation.

Args:

- `cell`: An instance of `RNNCell`.
- `inputs`: A length T list of inputs, each a `Tensor` of shape `[batch_size, input_size]`.
- `state_saver`: A state saver object with methods `state` and `save_state`.
- `state_name`: Python string or tuple of strings. The name to use with the state_saver. If the cell returns tuples of states (i.e., `cell.state_size` is a tuple) then `state_name` should be a tuple of strings having the same length as `cell.state_size`. Otherwise it should be a single string.
- `sequence_length`: (optional) An int32/int64 vector size [batch_size]. See the documentation for rnn() for more details about sequence_length.
- `scope`: VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (outputs, state) where: outputs is a length T list of outputs (one for each input) states is the final state

Raises:

- `TypeError`: If `cell` is not an instance of RNNCell.
- `ValueError`: If `inputs` is `None` or an empty list, or if the arity and type of `state_name` does not match that of `cell.state_size`.

`tf.nn.bidirectional_dynamic_rnn(cell_fw, cell_bw, inputs, sequence_length=None, initial_state_fw=None, initial_state_bw=None, dtype=None, parallel_iterations=None, swap_memory=False, time_major=False, scope=None)`

Creates a dynamic version of bidirectional recurrent neural network.

Similar to the unidirectional case above (rnn) but takes input and builds independent forward and backward RNNs. The input_size of forward and backward cell must match. The initial state for both directions is zero by default (but can be set optionally) and no intermediate states are ever returned -- the network is fully unrolled for the given (passed in) length(s) of the sequence(s) or completely unrolled if length(s) is not given.

Args:

- `cell_fw`: An instance of RNNCell, to be used for forward direction.
- `cell_bw`: An instance of RNNCell, to be used for backward direction.
- `inputs`: The RNN inputs. If time_major == False (default), this must be a tensor of shape: `[batch_size, max_time, input_size]`. If time_major == True, this must be a tensor of shape: `[max_time, batch_size, input_size]`. [batch_size, input_size].
- `sequence_length`: An int32/int64 vector, size `[batch_size]`, containing the actual lengths for each of the sequences.
- `initial_state_fw`: (optional) An initial state for the forward RNN. This must be a tensor of appropriate type and shape `[batch_size, cell_fw.state_size]`. If `cell_fw.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s] for s in cell_fw.state_size`.
- `initial_state_bw`: (optional) Same as for `initial_state_fw`, but using the corresponding properties of `cell_bw`.

- `dtype`: (optional) The data type for the initial states and expected output. Required if initial_states are not provided or RNN states have a heterogeneous dtype.

- `parallel_iterations`: (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer.

- `swap_memory`: Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.

- `time_major`: The shape format of the `inputs` and `outputs` Tensors. If true, these `Tensors` must be shaped `[max_time, batch_size, depth]`. If false, these `Tensors` must be shaped `[batch_size, max_time, depth]`. Using `time_major = True` is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.

- `dtype`: (optional) The data type for the initial state. Required if initial_state is not provided.

- `sequence_length`: An int32/int64 vector, size `[batch_size]`, containing the actual lengths for each of the sequences. either of the initial states are not provided.

- `scope`: VariableScope for the created subgraph; defaults to "BiRNN"

Returns:

A tuple (outputs, output_states) where:

- `outputs`: A tuple (output_fw, output_bw) containing the forward and the backward rnn output `Tensor`. If time_major == False (default), output_fw will be a `Tensor` shaped: `[batch_size, max_time, cell_fw.output_size]` and output_bw will be a `Tensor` shaped: `[batch_size, max_time, cell_bw.output_size]`. If time_major == True, output_fw will be a `Tensor` shaped: `[max_time, batch_size, cell_fw.output_size]` and output_bw will be a `Tensor` shaped: `[max_time, batch_size, cell_bw.output_size]`. It returns a tuple instead of a single concatenated `Tensor`, unlike in the `bidirectional_rnn`. If the concatenated one is preferred, the forward and backward outputs can be concatenated as `tf.concat(2, outputs)`.

- `output_states`: A tuple (output_state_fw, output_state_bw) containing the forward and the backward final states of bidirectional rnn.

Raises:

- `TypeError`: If `cell_fw` or `cell_bw` is not an instance of `RNNCell`.

`tf.nn.bidirectional_rnn(cell_fw, cell_bw, inputs, initial_state_fw=None, initial_state_bw=None, dtype=None, sequence_length=None, scope=None)`

Creates a bidirectional recurrent neural network.

Similar to the unidirectional case above (rnn) but takes input and builds independent forward and backward RNNs with the final forward and backward outputs depth-concatenated, such that the output will have the format [time][batch][cell_fw.output_size + cell_bw.output_size]. The input_size of forward and backward cell must match. The initial state for both directions is zero by default (but can be set optionally) and no intermediate states are ever returned -- the network is fully unrolled for the given (passed in) length(s) of the sequence(s) or completely unrolled if length(s) is not given.

Args:

- `cell_fw`: An instance of RNNCell, to be used for forward direction.
- `cell_bw`: An instance of RNNCell, to be used for backward direction.
- `inputs`: A length T list of inputs, each a tensor of shape [batch_size, input_size], or a nested tuple of such elements.
- `initial_state_fw`: (optional) An initial state for the forward RNN. This must be a tensor of appropriate type and shape `[batch_size, cell_fw.state_size]`. If `cell_fw.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s] for s in cell_fw.state_size`.
- `initial_state_bw`: (optional) Same as for `initial_state_fw`, but using the corresponding properties of `cell_bw`.
- `dtype`: (optional) The data type for the initial state. Required if either of the initial states are not provided.
- `sequence_length`: (optional) An int32/int64 vector, size `[batch_size]`, containing the actual lengths for each of the sequences.
- `scope`: VariableScope for the created subgraph; defaults to "BiRNN"

Returns:

A tuple (outputs, output_state_fw, output_state_bw) where: outputs is a length `T` list of outputs (one for each input), which are depth-concatenated forward and backward outputs. output_state_fw is the final state of the forward rnn. output_state_bw is the final state of the backward rnn.

Raises:

- `TypeError`: If `cell_fw` or `cell_bw` is not an instance of `RNNCell`.
- `ValueError`: If inputs is None or an empty list.

`tf.nn.raw_rnn(cell, loop_fn, parallel_iterations=None, swap_memory=False, scope=None)`

Creates an `RNN` specified by RNNCell `cell` and loop function `loop_fn`.

NOTE: This method is still in testing, and the API may change.

This function is a more primitive version of `dynamic_rnn` that provides more direct access to the inputs each iteration. It also provides more control over when to start and finish reading the sequence, and what to emit for the output.

For example, it can be used to implement the dynamic decoder of a seq2seq model.

Instead of working with `Tensor` objects, most operations work with `TensorArray` objects directly.

The operation of `raw_rnn`, in pseudo-code, is basically the following:

```
time = tf.constant(0, dtype=tf.int32)
(finished, next_input, initial_state, _, loop_state) = loop_fn(
    time=time, cell_output=None, cell_state=None, loop_state=None)
emit_ta = TensorArray(dynamic_size=True, dtype=initial_state.dtype)
state = initial_state
while not all(finished):
  (output, cell_state) = cell(next_input, state)
  (next_finished, next_input, next_state, emit, loop_state) = loop_fn(
      time=time + 1, cell_output=output, cell_state=cell_state,
      loop_state=loop_state)
  # Emit zeros and copy forward state for minibatch entries that are finished.
  state = tf.select(finished, state, next_state)
  emit = tf.select(finished, tf.zeros_like(emit), emit)
  emit_ta = emit_ta.write(time, emit)
  # If any new minibatch entries are marked as finished, mark these
  finished = tf.logical_or(finished, next_finished)
  time += 1
return (emit_ta, state, loop_state)
```

with the additional properties that output and state may be (possibly nested) tuples, as determined by `cell.output_size` and `cell.state_size`, and as a result the final `state` and `emit_ta` may themselves be tuples.

A simple implementation of `dynamic_rnn` via `raw_rnn` looks like this:

```
inputs = tf.placeholder(shape=(max_time, batch_size, input_depth),
                        dtype=tf.float32)
sequence_length = tf.placeholder(shape=(batch_size,), dtype=tf.int32)
inputs_ta = tf.TensorArray(dtype=tf.float32, size=max_time)
inputs_ta = inputs_ta.unpack(inputs)

cell = tf.nn.rnn_cell.LSTMCell(num_units)

def loop_fn(time, cell_output, cell_state, loop_state):
  emit_output = cell_output  # == None for time == 0
  if cell_output is None:  # time == 0
    next_cell_state = cell.zero_state(batch_size, tf.float32)
  else:
    next_cell_state = cell_state
  elements_finished = (time >= sequence_length)
  finished = tf.reduce_all(elements_finished)
  next_input = tf.cond(
      finished,
      lambda: tf.zeros([batch_size, input_depth], dtype=tf.float32),
      lambda: inputs_ta.read(time))
  next_loop_state = None
  return (elements_finished, next_input, next_cell_state,
          emit_output, next_loop_state)

outputs_ta, final_state, _ = raw_rnn(cell, loop_fn)
outputs = outputs_ta.pack()
```

Args:

- `cell`: An instance of RNNCell.

  `loop_fn`: A callable that takes inputs `(time, cell_output, cell_state, loop_state)` and returns the tuple `(finished, next_input, next_cell_state, emit_output, next_loop_state)`. Here `time` is an int32 scalar `Tensor`, `cell_output` is a `Tensor` or (possibly nested) tuple of tensors as determined by `cell.output_size`, and `cell_state` is a `Tensor` or (possibly nested) tuple of tensors, as determined by the `loop_fn` on its first call (and should match `cell.state_size`). The outputs are: `finished`, a boolean `Tensor` of shape `[batch_size]`, `next_input`: the next input to feed to `cell`, `next_cell_state`: the next state to feed to `cell`, and `emit_output`: the output to store for this iteration.

Note that `emit_output` should be a `Tensor` or (possibly nested) tuple of tensors with shapes and structure matching `cell.output_size` and `cell_output` above. The parameter `cell_state` and output `next_cell_state` may be either a single or (possibly nested) tuple of tensors. The parameter `loop_state` and output `next_loop_state` may be either a single or (possibly nested) tuple of `Tensor` and `TensorArray` objects. This last parameter may be ignored by `loop_fn` and the return value may be `None`. If it is not `None`, then the `loop_state` will be propagated through the RNN loop, for use purely by `loop_fn` to keep track of its own state. The `next_loop_state` parameter returned may be `None`.

The first call to `loop_fn` will be `time = 0`, `cell_output = None`, `cell_state = None`, and `loop_state = None`. For this call: The `next_cell_state` value should be the value with which to initialize the cell's state. It may be a final state from a previous RNN or it may be the output of `cell.zero_state()`. It should be a (possibly nested) tuple structure of tensors. If `cell.state_size` is an integer, this must be a `Tensor` of appropriate type and shape `[batch_size, cell.state_size]`. If `cell.state_size` is a `TensorShape`, this must be a `Tensor` of appropriate type and shape `[batch_size] + cell.state_size`. If `cell.state_size` is a (possibly nested) tuple of ints or `TensorShape`, this will be a tuple having the corresponding shapes. The `emit_output` value may be either `None` or a (possibly nested) tuple structure of tensors, e.g., `(tf.zeros(shape_0, dtype=dtype_0), tf.zeros(shape_1, dtype=dtype_1))`. If this first `emit_output` return value is `None`, then the `emit_ta` result of `raw_rnn` will have the same structure and dtypes as `cell.output_size`. Otherwise `emit_ta` will have the same structure, shapes (prepended with a `batch_size` dimension), and dtypes as `emit_output`. The actual values returned for `emit_output` at this initializing call are ignored. Note, this emit structure must be consistent across all time steps.

`parallel_iterations`: (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values >> 1 use more memory but take less time, while smaller values use less memory but computations take longer.

`swap_memory`: Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.

`scope`: VariableScope for the created subgraph; defaults to "RNN".

Returns:

A tuple `(emit_ta, final_state, final_loop_state)` where:

`emit_ta`: The RNN output `TensorArray`. If `loop_fn` returns a (possibly nested) set of Tensors for `emit_output` during initialization, (inputs `time = 0`, `cell_output = None`, and `loop_state = None`), then `emit_ta` will have the same structure, dtypes, and shapes as `emit_output` instead. If `loop_fn` returns `emit_output = None` during this call, the structure of `cell.output_size` is used: If `cell.output_size` is a (possibly nested) tuple of integers or `TensorShape` objects, then `emit_ta` will be a tuple having the same structure as `cell.output_size`, containing TensorArrays whose elements' shapes correspond to the shape data in `cell.output_size`.

`final_state`: The final cell state. If `cell.state_size` is an int, this will be shaped `[batch_size, cell.state_size]`. If it is a `TensorShape`, this will be shaped `[batch_size] + cell.state_size`. If it is a (possibly nested) tuple of ints or `TensorShape`, this will be a tuple having the corresponding shapes.

`final_loop_state`: The final loop state as returned by `loop_fn`.

Raises:

- `TypeError`: If `cell` is not an instance of RNNCell, or `loop_fn` is not a `callable`.

Conectionist Temporal Classification (CTC)

`tf.nn.ctc_loss(inputs, labels, sequence_length, preprocess_collapse_repeated=False, ctc_merge_repeated=True, time_major=True)`

Computes the CTC (Connectionist Temporal Classification) Loss.

This op implements the CTC loss as presented in the article:

A. Graves, S. Fernandez, F. Gomez, J. Schmidhuber. Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks. ICML 2006, Pittsburgh, USA, pp. 369-376.

http://www.cs.toronto.edu/~graves/icml_2006.pdf

Input requirements:

```
sequence_length(b) <= time for all b
```

```
max(labels.indices(labels.indices[:, 1] == b, 2))
  <= sequence_length(b) for all b.
```

Notes:

This class performs the softmax operation for you, so inputs should be e.g. linear projections of outputs by an LSTM.

The `inputs` Tensor's innermost dimension size, `num_classes`, represents `num_labels + 1` classes, where num_labels is the number of true labels, and the largest value `(num_classes - 1)` is reserved for the blank label.

For example, for a vocabulary containing 3 labels `[a, b, c]`, `num_classes = 4` and the labels indexing is `{a: 0, b: 1, c: 2, blank: 3}`.

Regarding the arguments `preprocess_collapse_repeated` and `ctc_merge_repeated`:

If `preprocess_collapse_repeated` is True, then a preprocessing step runs before loss calculation, wherein repeated labels passed to the loss are merged into single labels. This is useful if the training labels come from, e.g., forced alignments and therefore have unnecessary repetitions.

If `ctc_merge_repeated` is set False, then deep within the CTC calculation, repeated non-blank labels will not be merged and are interpreted as individual labels. This is a simplified (non-standard) version of CTC.

Here is a table of the (roughly) expected first order behavior:

> `preprocess_collapse_repeated=False`, `ctc_merge_repeated=True`
>
> Classical CTC behavior: Outputs true repeated classes with blanks in between, and can also output repeated classes with no blanks in between that need to be collapsed by the decoder.
>
> `preprocess_collapse_repeated=True`, `ctc_merge_repeated=False`
>
> Never learns to output repeated classes, as they are collapsed in the input labels before training.
>
> `preprocess_collapse_repeated=False`, `ctc_merge_repeated=False`
>
> Outputs repeated classes with blanks in between, but generally does not require the decoder to collapse/merge repeated classes.
>
> `preprocess_collapse_repeated=True`, `ctc_merge_repeated=True`

Untested. Very likely will not learn to output repeated classes.

Args:

- `inputs`: 3-D `float` `Tensor`. If time_major == False, this will be a `Tensor` shaped: `[batch_size x max_time x num_classes]`. If time_major == True (default), this will be a `Tensor` shaped: `[max_time x batch_size x num_classes]`. The logits.
- `labels`: An `int32` `SparseTensor`. `labels.indices[i, :] == [b, t]` means `labels.values[i]` stores the id for (batch b, time t). `labels.values[i]` must take on values in `[0, num_labels)`. See `core/ops/ctc_ops.cc` for more details.
- `sequence_length`: 1-D `int32` vector, size `[batch_size]`. The sequence lengths.
- `preprocess_collapse_repeated`: Boolean. Default: False. If True, repeated labels are collapsed prior to the CTC calculation.
- `ctc_merge_repeated`: Boolean. Default: True.
- `time_major`: The shape format of the `inputs` Tensors. If True, these `Tensors` must be shaped `[max_time, batch_size, num_classes]`. If False, these `Tensors` must be shaped `[batch_size, max_time, num_classes]`. Using `time_major = True` (default) is a bit more efficient because it avoids transposes at the beginning of the ctc_loss calculation. However, most TensorFlow data is batch-major, so by this function also accepts inputs in batch-major form.

Returns:

A 1-D `float` `Tensor`, size `[batch]`, containing the negative log probabilities.

Raises:

- `TypeError`: if labels is not a `SparseTensor`.

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

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

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

Args:

- `inputs`: 3-D `float` `Tensor` sized `[max_time x batch_size x 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, log_probabilities)` where

- `decoded`: A single-element list. `decoded[0]` is an `SparseTensor` containing the decoded outputs s.t.: `decoded.indices`: Indices matrix `(total_decoded_outputs x 2)`. The rows store: `[batch, time]`. `decoded.values`: Values vector, size `(total_decoded_outputs)`. The vector stores the decoded classes. `decoded.shape`: Shape vector, size `(2)`. The shape values are: `[batch_size, max_decoded_length]`
- `log_probability`: A `float` matrix `(batch_size x 1)` containing sequence log-probabilities.

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

Performs beam search decoding on the logits given in input.

Note The `ctc_greedy_decoder` is a special case of the `ctc_beam_search_decoder` with `top_paths=1` (but that decoder is faster for this special case).

If `merge_repeated` is `True`, merge repeated classes in the output beams. This means that if consecutive entries in a beam are the same, only the first of these is emitted. That is, when the top path is `A B B B B`, the return value is:

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

Args:

- `inputs`: 3-D `float` `Tensor`, size `[max_time x batch_size x 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).
- `merge_repeated`: Boolean. Default: True.

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] x 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].shape`: Shape vector, size `(2)`. The shape values are: `[batch_size, max_decoded_length[j]]`.
- `log_probability`: A `float` matrix `(batch_size x top_paths)` containing sequence log-probabilities.

Evaluation

The evaluation ops are useful for measuring the performance of a network. Since they are nondifferentiable, they are typically used at evaluation time.

`tf.nn.top_k(input, k=1, sorted=True, name=None)`

Finds values and indices of the `k` largest entries for the last dimension.

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.nn.in_top_k(predictions, targets, k, name=None)`

Says whether the targets are in the top `K` predictions.

This outputs a `batch_size` bool array, an entry `out[i]` is `true` if the prediction for the target class is 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

$\boldsymbol{predictions_i}$ be the predictions for all classes for example `i`, $\boldsymbol{targets_i}$ be the target class for example `i`, $\boldsymbol{out_i}$ be the output for example `i`,

$$out_i = predictions_{i,targets_i} \in TopKIncludingTies(predictions_i)$$

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`.

Candidate Sampling

Do you want to train a multiclass or multilabel model with thousands or millions of output classes (for example, a language model with a large vocabulary)? Training with a full Softmax is slow in this case, since all of the classes are evaluated for every training example. Candidate Sampling training algorithms can speed up your step times by only considering a small randomly-chosen subset of contrastive classes (called candidates) for each batch of training examples.

See our Candidate Sampling Algorithms Reference

Sampled Loss Functions

TensorFlow provides the following sampled loss functions for faster training.

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

Computes and returns the noise-contrastive estimation training loss.

See Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. Also see our Candidate Sampling Algorithms Reference

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 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.
- `inputs`: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- `labels`: A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `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. 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.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(weights) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name`: A name for the operation (optional).

Returns:

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

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

Computes and returns the sampled softmax training loss.

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.

At inference time, you can compute full softmax probabilities with the expression `tf.nn.softmax(tf.matmul(inputs, tf.transpose(weights)) + biases)`.

See our Candidate Sampling Algorithms Reference

Also see Section 3 of Jean et al., 2014 (pdf) for the math.

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.
- `inputs`: A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- `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`.
- `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.
- `partition_strategy`: A string specifying the partitioning strategy, relevant if `len(weights) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name`: A name for the operation (optional).

Returns:

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

Candidate Samplers

TensorFlow provides the following samplers for randomly sampling candidate classes when using one of the sampled loss functions above.

`tf.nn.uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`

Samples a set of classes using a uniform base distribution.

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 per batch.
- `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.nn.log_uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`

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

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 per batch.
- `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.nn.learned_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)`

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

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 per batch.
- `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.nn.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)`

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

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 per batch.

- `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`.

Miscellaneous candidate sampling utilities

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

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

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`.

Other Functions and Classes

`tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)`

Batch normalization.

As described in http://arxiv.org/abs/1502.03167. Normalizes a tensor by `mean` and `variance`, and applies (optionally) a `scale` $\backslash\gamma$ to it, as well as an `offset` $\backslash \boldsymbol{beta}$:

$$\backslash \boldsymbol{frac}\gamma(x-\mu)\sigma +$$
$$\boldsymbol{beta}$$

`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.

Args:

- `x`: Input `Tensor` of arbitrary dimensionality.
- `mean`: A mean `Tensor`.
- `variance`: A variance `Tensor`.

- `offset`: An offset `Tensor`, often denoted \$beta$ 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.depthwise_conv2d_native(input, filter, strides, padding, name=None)`

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

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, 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. Thus, the output has `in_channels * channel_multiplier` channels.

for k in 0..in_channels-1 for q in 0..channel_multiplier-1 output[b, i, j, k * channel_multiplier + q] = sum_{di, dj} input[b, strides[1] * i + di, strides[2] * j + dj, k] * filter[di, dj, k, q]

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: `float32`, `float64`.
- `filter`: A `Tensor`. Must have the same type as `input`.

* `strides`: A list of `ints`. 1-D of length 4. The stride of the sliding window for each dimension of `input`.
* `padding`: A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
* `name`: A name for the operation (optional).

Returns:

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