新手入门完整教程进阶指南
API中文手册精华文章TF社区

maximum

- **3.** 进阶指南
  - **3.5.** 读取数据
- **6.** API DOC
  - **6.3.** Python API
    - **6.3.5.** Framework
    - **6.3.6.** Image
    - **6.3.7.** Index
    - **6.3.8.** Io Ops
    - **6.3.9.** Math Ops
    - **6.3.10.** Nn
    - **6.3.13.** State Ops
    - **6.3.14.** Train

Facebook Google+ Twitter Weibo Instapaper
A A
Serif Sans
White Sepia Night

TensorFlow 官方文档中文版

Neural Network

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

Contents

    - tf.nn.avg_pool(value, ksize, strides, padding, name=None)
    - tf.nn.max_pool(value, ksize, strides, padding, name=None)
    - tf.nn.max_pool_with_argmax(input, ksize, strides, padding, Targmax=None, 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.moments(x, axes, name=None)
- Losses
    - tf.nn.l2_loss(t, name=None)
- Classification
    - tf.nn.sigmoid_cross_entropy_with_logits(logits, targets, name=None)
    - tf.nn.softmax(logits, name=None)
    - tf.nn.softmax_cross_entropy_with_logits(logits, labels, name=None)
- Embeddings
    - tf.nn.embedding_lookup(params, ids, name=None)
- Evaluation
    - tf.nn.top_k(input, k, 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, 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, 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=0.0, 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)

Activation Functions

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities (`sigmoid`, `tanh`, and `softplus`), continuous but not everywhere differentiable functions (`relu`, `relu6`, 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`.
- **`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.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`.
- **`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 Python float. 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, 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`, or `complex64`.
- **`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.
- **`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 `float`, `double`, `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 with type `float`, `double`, `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`.

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, the spatial semantics of the convolution ops are as follows. If the 4-D `input` has shape `[batch, in_height, in_width, ...]` and the 4-D `filter` has shape `[filter_height, filter_width, ...]`, then

```
shape(output) = [batch,
                 (in_height - filter_height + 1) / strides[1],
                 (in_width - filter_width + 1) / strides[2],
                 ...]

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

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.

In the formula for `shape(output)`, the rounding direction depends on padding:

- `padding = 'SAME'`: Round down (only full size windows are considered).
- `padding = 'VALID'`: Round up (partial windows are included).

`tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=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 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,

```
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: `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.
- **`use_cudnn_on_gpu`**: An optional `bool`. Defaults to `True`.
- **`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.
- **`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.
- **`name`**: A name for this operation (optional).

Returns:

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

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])
```

for each tuple of indices `i`. The output shape is

```
shape(output) = (shape(value) - ksize + 1) / strides
```

where the rounding direction depends on padding:

- `padding = 'SAME'`: Round down (only full size windows are considered).
- `padding = 'VALID'`: Round up (partial windows are included).

`tf.nn.avg_pool(value, ksize, strides, padding, 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.
- **`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, name=None)`

Performs the max pooling on the input.

Args:

- **`value`**: A 4-D `Tensor` with shape `[batch, height, width, channels]` and type `float32`, `float64`, `qint8`, `quint8`, `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.
- **`name`**: Optional name for the operation.

Returns:

A `Tensor` with the same type as `value`. 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` of type `float32`. 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` of type `float32`. The max pooled output tensor.
- **`argmax`**: A `Tensor` of type `Targmax`. 4-D. The flattened indices of the max values chosen for each output.

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.
- **`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)] (http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks).

Args:

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

Returns:

A `Tensor` of type `float32`.

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

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.

For so-called "global normalization" needed for convolutional filters pass `axes=[0, 1, 2]` (batch, height, width). For batch normalization pass `axes=[0]` (batch).

Args:

- **`x`**: A `Tensor`.
- **`axes`**: array of ints. Axes along which to compute mean and variance.
- **`name`**: Name used to scope the operations that compute the moments.

Returns:

Two `Tensors`: `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`, `int16`, `int8`, `complex64`, `qint8`, `quint8`, `qint32`. 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.

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

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

To ensure stability and avoid overflow, the implementation uses

```
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.

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

Computes softmax activations.

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

```
softmax[i, j] = exp(logits[i, j]) / sum(exp(logits[i]))
```

Args:

- **`logits`**: A `Tensor`. Must be one of the following types: `float32`, `float64`. 2-D with shape `[batch_size, num_classes]`.
- **`name`**: A name for the operation (optional).

Returns:

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

`tf.nn.softmax_cross_entropy_with_logits(logits, labels, 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.

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 `float32` or `float64`).

Args:

- **`logits`**: Unscaled log probabilities.
- **`labels`**: Each row `labels[i]` must be a valid probability distribution.
- **`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.

Embeddings

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

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

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` by computing `p = id % len(params)`, and is then used to look up the slice `params[p][id // len (params), ...]`.

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

Args:

- **`params`**: A list of tensors with the same shape and type.
- **`ids`**: A `Tensor` with type `int32` containing the ids to be looked up in `params`.
- **`name`**: A name for the operation (optional).

Returns:

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

Raises:

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

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, name=None)`

Returns the values and indices of the k largest elements for each row.

$values_{i,j}$ represents the j-th largest element in $input_i$.

$indices_{i,j}$ gives the column index of the corresponding element, such that $input_{i,indices_{i,j}} = values_{i,j}$. If two elements are equal, the lower-index element appears first.

Args:

- **`input`**: A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`. A batch_size x classes tensor
- **`k`**: An `int` that is `>= 1`. Number of top elements to look for within each row
- **`name`**: A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (values, indices).

- **`values`**: A `Tensor`. Has the same type as `input`. A batch_size x k tensor with the k largest elements for each row, sorted in descending order
- **`indices`**: A `Tensor` of type `int32`. A batch_size x k tensor with the index of each value within each row

---

`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

$predictions_i$ be the predictions for all classes for example i, $targets_i$ be the target class for example i, 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` of type `int32`. 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] (../../extras/candidate_sampling.pdf)

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, name='nce_loss')`

Computes and returns the noise-contrastive estimation training loss.

See [Noise-contrastive estimation: A new estimation principle for unnormalized statistical models] (http://www.jmlr.org/proceedings/papers/v9/gutmann10a/gutmann10a.pdf). Also see our [Candidate Sampling Algorithms Reference] (http://www.tensorflow.org/extras/candidate_sampling.pdf)

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]. The 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 LogUniformCandidateSampler)
- **`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] (http://www.tensorflow.org/extras/candidate_sampling.pdf). Default is False.
- **`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, 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, weights) + biases)`.

See our [Candidate Sampling Algorithms Reference] (http://www.tensorflow.org/extras/candidate_sampling.pdf)

Also see Section 3 of http://arxiv.org/abs/1412.2007 for the math.

Args:

- **`weights`**: A `Tensor` of shape [num_classes, dim]. The 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 LogUniformCandidateSampler)
- **`remove_accidental_hits`**: A `bool`. whether to remove "accidental hits" where a sampled class equals one of the target classes. Default is True.
- **`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=0.0, 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 ids of positions 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`.