cntk.layers.layers module

Blocks in the network that are used layer-like, i.e. layered on top of each other e.g. a fully connected layer with non-linearity.

Activation(*activation=identity, name=''*) [source]

Layer factory function to create an activation layer. Activation functions can be used directly in CNTK, so there is no difference between `y = relu(x)` and `y = Activation(relu)(x)`. This layer is useful if one wants to configure the activation function with `default_options`, or when its invocation should be named.

Example

```
>>> model = Dense(500) >> Activation(C.relu) >> Dense(10)
>>> # is the same as
>>> model = Dense(500) >> C.relu >> Dense(10)
>>> # and also the same as
>>> model = Dense(500, activation=C.relu) >> Dense(10)
```

Parameters:
- **activation** (`Function`, defaults to *identity*) – function to apply at the end, e.g. *relu*
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

AveragePooling(*filter_shape, strides=1, pad=False, name=''*) [source]

Layer factory function to create an average-pooling layer.

Like `Convolution()`, `AveragePooling()` processes items arranged on an N-dimensional grid, such as an image. Typically, each item is a vector. For each item, average-pooling computes the element-wise mean over a window ("receptive field") of items surrounding the item's position on the grid.

The size (spatial extent) of the receptive field is given by `filter_shape`. E.g. for 2D pooling, `filter_shape` should be a tuple of two integers, such as *(5,5)*.

Example

```
>>> f = AveragePooling((3,3), strides=2)  # reduce dimensionality by 2, pooling over
windows of 3x3
>>> h = C.input_variable((32,240,320))  # e.g. 32-dim feature map
>>> hp = f(h)
>>> hp.shape  # spatial dimension has been halved due to stride, and lost one due to 3x3
window without padding
    (32, 119, 159)
```

```
>>> f = AveragePooling((2,2), strides=2)
>>> f.update_signature((1,4,4))
>>> im = np.array([[[3, 5, 2, 6], [4, 2, 8, 3], [1, 6, 4, 7], [7, 3, 5, 9]]])  # a 4x4
image (feature-map depth 1 for simplicity)
>>> im
    array([[[3, 5, 2, 6],
            [4, 2, 8, 3],
            [1, 6, 4, 7],
            [7, 3, 5, 9]]])
>>> f([[im]])  # due to strides=2, this computes the averages of each 2x2 sub-block
    array([[[[ 3.5 ,  4.75],
             [ 4.25,  6.25]]]], dtype=float32)
```

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride (increment when sliding over the input). Use a *tuple* to specify a per-axis value.
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the pooling operation will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, pooling will be applied to all input positions, and positions outside the valid region will be excluded from the averaging. Use a *tuple* to specify a per-axis value.
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the average-pooling operation to it

Return type: cntk.ops.functions.Function

BatchNormalization(*map_rank=None, init_scale=1, normalization_time_constant=5000, blend_time_constant=0, epsilon=0.00001, use_cntk_engine=False, name=''*) [source]

Layer factory function to create a batch-normalization layer.

Batch normalization applies this formula to every input element (element-wise): `y = (x - batch_mean) / (batch_stddev + epsilon) * scale + bias` where `batch_mean` and `batch_stddev` are estimated on the minibatch and `scale` and `bias` are learned parameters.

During operation, this layer also estimates an aggregate running mean and standard deviation for use in inference.

A `BatchNormalization` layer instance owns its learnable parameter tensors and exposes them as attributes `.scale` and `.bias`. The aggregate estimates are exposed as attributes `aggregate_mean`, `aggregate_variance`, and `aggregate_count`.

Example

```
>>> # BatchNorm on an image with spatial pooling
>>> f = BatchNormalization(map_rank=1)
>>> f.update_signature((3,480,640))
>>> f.bias.shape, f.scale.shape  # due to spatial pooling (map_rank=1), there are only 3
biases and scales, shared across all pixel positions
    ((3,), (3,))
```

Parameters:

- **map_rank** (1 or `None`) – passing 1 means spatially-pooled batch-normalization, where normalization values will be tied across all pixel positions; while `None` will normalize all elements of the input tensor independently
- **init_scale** (*float, default 1*) – initial value for the `scale` parameter
- **normalization_time_constant** (*int, default 5000*) – time constant for smoothing the batch statistics in order to compute aggregate estimates for inference.
- **epsilon** (*float, default 0.00001*) – epsilon added to the variance to avoid division by 0
- **use_cntk_engine** (bool, default `False`) – if `True` then use CNTK's own engine instead of NVidia's.
- **name** (*str, optional*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

Convolution(*filter_shape, num_filters=None, sequential=False, activation=identity, init=glorot_uniform(), pad=False, strides=1, sharing=True, bias=True, init_bias=0, reduction_rank=1, transpose_weight=False, max_temp_mem_size_in_samples=0, op_name='Convolution', name=''*)
[source]

Layer factory function to create a convolution layer.

This implements a convolution operation over items arranged on an N-dimensional grid, such as pixels in an image. Typically, each item is a vector (e.g. pixel: R,G,B), and the result is, in turn, a vector. The item-grid dimensions are referred to as the *spatial* dimensions (e.g. dimensions of an image), while the vector dimension of the individual items is often called *feature-map depth*.

For each item, convolution gathers a window ("receptive field") of items surrounding the item's position on the grid, and applies a little fully-connected network to it (the same little network is applied to all item positions). The size (spatial extent) of the receptive field

is given by `filter_shape`. E.g. to specify a 2D convolution, `filter_shape` should be a tuple of two integers, such as *(5,5)*; an example for a 3D convolution (e.g. video or an MRI scan) would be `filter_shape=(3,3,3)`; while for a 1D convolution (e.g. audio or text), `filter_shape` has one element, such as (3,) or just 3.

The dimension of the input items (input feature-map depth) is not to be specified. It is known from the input. The dimension of the output items (output feature-map depth) generated for each item position is given by `num_filters`.

If the input is a sequence, the sequence elements are by default treated independently. To convolve along the sequence dimension as well, pass `sequential=True`. This is useful for variable-length inputs, such as video or natural-language processing (word n-grams). Note, however, that convolution does not support sparse inputs.

Both input and output items can be scalars intead of vectors. For scalar-valued input items, such as pixels on a black-and-white image, or samples of an audio clip, specify `reduction_rank=0`. If the output items are scalar, pass `num_filters=()` or `None`.

A `Convolution` instance owns its weight parameter tensors *W* and *b*, and exposes them as an attributes `.W` and `.b`. The weights will have the shape `(num_filters, input_feature_map_depth, *filter_shape)`

Example

```
>>> # 2D convolution of 5x4 receptive field with output feature-map depth 128:
>>> f = Convolution((5,4), 128, activation=C.relu)
>>> x = C.input_variable((3,480,640))  # 3-channel color image
>>> h = f(x)
>>> h.shape
    (128, 476, 637)
>>> f.W.shape  # will have the form (num_filters, input_depth, *filter_shape)
    (128, 3, 5, 4)
```

```
>>> # 2D convolution over a one-channel black-and-white image, padding, and stride 2 along
width dimension
>>> f = Convolution((3,3), 128, reduction_rank=0, pad=True, strides=(1,2),
activation=C.relu)
>>> x = C.input_variable((480,640))
>>> h = f(x)
>>> h.shape
    (128, 480, 320)
>>> f.W.shape
    (128, 1, 3, 3)
```

```
>>> # 3D convolution along dynamic axis over a sequence of 2D color images
>>> from cntk.layers.typing import Sequence, Tensor
>>> f = Convolution((2,5,4), 128, sequential=True, activation=C.relu) # over 2 consecutive
frames
>>> x = C.input_variable(**Sequence[Tensor[3,480,640]])  # a variable-length video of
640x480 RGB images
>>> h = f(x)
>>> h.shape   # this is the shape per video frame: 637x476 activation vectors of length 128
each
    (128, 476, 637)
>>> f.W.shape # (output featuer map depth, input depth, and the three filter dimensions)
    (128, 3, 2, 5, 4)
```

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **num_filters** (int, defaults to *None*) – number of filters (output feature-map depth), or `()` to denote scalar output items (output shape will have no depth axis).
- **sequential** (bool, defaults to *False*) – if *True*, also convolve along the dynamic axis. `filter_shape[0]` corresponds to dynamic axis.
- **activation** (`Function` , defaults to *identity*) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or NumPy array or `cntk.initializer` , defaults to `glorot_uniform()`) – initial value of weights *W*
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the filter will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, the filter will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride of the convolution (increment when sliding the filter over the input). Use a *tuple* to specify a per-axis value.
- **sharing** (bool, defaults to *True*) – When *True*, every position uses the same Convolution kernel. When *False*, you can have a different Convolution kernel per position, but *False* is not supported.
- **bias** (bool, optional, defaults to *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer` , defaults to 0) – initial value of weights *b*
- **reduction_rank** (*int*, defaults to 1) – set to 0 if input items are scalars (input has no depth axis), e.g. an audio signal or a black-and-white image that is stored with tensor shape (H,W) instead of (1,H,W)
- **transpose_weight** (bool, defaults to *False*) – When this is *True* this is convolution, otherwise this is correlation (which is common for most toolkits)
- **max_temp_mem_size_in_samples** (*int, defaults to 0*) – Limits the amount of memory for intermediate convolution results. A value of 0 means, memory is automatically managed.
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the convolution operation to it

Return type: cntk.ops.functions.Function

Convolution1D(*filter_shape, num_filters=None, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, reduction_rank=1, name=''*) [source]

Layer factory function to create a 1D convolution layer with optional non-linearity. Same as *Convolution()* except that filter_shape is verified to be 1-dimensional. See *Convolution()* for extensive documentation.

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **num_filters** (int, defaults to *None*) – number of filters (output feature-map depth), or `()` to denote scalar output items (output shape will have no depth axis).
- **activation** (`Function`, defaults to *identity*) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or NumPy array or `cntk.initializer`, defaults to `glorot_uniform()`) – initial value of weights *W*
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the filter will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, the filter will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride of the convolution (increment when sliding the filter over the input). Use a *tuple* to specify a per-axis value.
- **bias** (bool, defaults to *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer`, defaults to 0) – initial value of weights *b*
- **reduction_rank** (*int*, defaults to 1) – set to 0 if input items are scalars (input has no depth axis), e.g. an audio signal or a black-and-white image that is stored with tensor shape (H,W) instead of (1,H,W)
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the convolution operation to it

Return type: cntk.ops.functions.Function

Convolution2D(*filter_shape, num_filters=None, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, reduction_rank=1, name=''*) [source]

Layer factory function to create a 2D convolution layer with optional non-linearity. Same as *Convolution()* except that filter_shape is verified to be 2-dimensional. See *Convolution()* for extensive documentation.

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **num_filters** (int, defaults to *None*) – number of filters (output feature-map depth), or `()` to denote scalar output items (output shape will have no depth axis).
- **activation** (`Function`, defaults to *identity*) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or NumPy array or `cntk.initializer`, defaults to `glorot_uniform()`) – initial value of weights *W*
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the filter will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, the filter will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride of the convolution (increment when sliding the filter over the input). Use a *tuple* to specify a per-axis value.
- **bias** (bool, defaults to *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer`, defaults to 0) – initial value of weights *b*
- **reduction_rank** (*int*, defaults to 1) – set to 0 if input items are scalars (input has no depth axis), e.g. an audio signal or a black-and-white image that is stored with tensor shape (H,W) instead of (1,H,W)
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the convolution operation to it

Return type: cntk.ops.functions.Function

Convolution3D(*filter_shape, num_filters=None, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, reduction_rank=1, name=''*) [source]

Layer factory function to create a 3D convolution layer with optional non-linearity. Same as *Convolution()* except that filter_shape is verified to be 3-dimensional. See *Convolution()* for extensive documentation.

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **num_filters** (int, defaults to *None*) – number of filters (output feature-map depth), or `()` to denote scalar output items (output shape will have no depth axis).
- **activation** (`Function`, defaults to *identity*) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or NumPy array or `cntk.initializer`, defaults to `glorot_uniform()`) – initial value of weights *W*
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the filter will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, the filter will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride of the convolution (increment when sliding the filter over the input). Use a *tuple* to specify a per-axis value.
- **bias** (bool, defaults to *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer`, defaults to 0) – initial value of weights *b*
- **reduction_rank** (*int*, defaults to 1) – set to 0 if input items are scalars (input has no depth axis), e.g. an audio signal or a black-and-white image that is stored with tensor shape (H,W) instead of (1,H,W)
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the convolution operation to it

Return type: cntk.ops.functions.Function

`ConvolutionTranspose`(*filter_shape, num_filters, activation=identity, init=glorot_uniform(), pad=False, strides=1, sharing=True, bias=True, init_bias=0, output_shape=None, reduction_rank=1, max_temp_mem_size_in_samples=0, name=''*) [source]

Layer factory function to create a convolution transpose layer.

This implements a convolution_transpose operation over items arranged on an N-dimensional grid, such as pixels in an image. Typically, each item is a vector (e.g. pixel: R,G,B), and the result is, in turn, a vector. The item-grid dimensions are referred to as the *spatial* dimensions (e.g. dimensions of an image), while the vector dimensions of the individual items are often called *feature-map depth*.

Convolution transpose is also known as `fractionally strided convolutional layers`, or, `deconvolution`. This operation is used in image and language processing applications. It supports arbitrary dimensions, strides, and padding.

The forward and backward computation of convolution transpose is the inverse of convolution. That is, during forward pass the input layer's items are spread into the output same as the backward spread of gradients in convolution. The backward pass, on the other hand, performs a convolution same as the forward pass of convolution.

The size (spatial extent) of the receptive field for convolution transpose is given by `filter_shape`. E.g. to specify a 2D convolution transpose, `filter_shape` should be a tuple of two integers, such as *(5,5)*; an example for a 3D convolution transpose (e.g. video or an MRI scan) would be `filter_shape=(3,3,3)`; while for a 1D convolution transpose (e.g. audio or text), `filter_shape` has one element, such as (3,).

The dimension of the input items (feature-map depth) is not specified, but known from the input. The dimension of the output items generated for each item position is given by `num_filters`.

A `ConvolutionTranspose` instance owns its weight parameter tensors *W* and *b*, and exposes them as an attributes `.W` and `.b`. The weights will have the shape `(input_feature_map_depth, num_filters, *filter_shape)`.

Example

```
>>> # 2D convolution transpose of 3x4 receptive field with output feature-map depth 128:
>>> f = ConvolutionTranspose((3,4), 128, activation=C.relu)
>>> x = C.input_variable((3,480,640))  # 3-channel color image
>>> h = f(x)
>>> h.shape
    (128, 482, 643)
>>> f.W.shape  # will have the form (input_depth, num_filters, *filter_shape)
    (3, 128, 3, 4)
```

Parameters:

- **filter_shape** (*int* or tuple of *int*s) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **num_filters** (*int*) – number of filters (output feature-map depth), or `()` to denote scalar output items (output shape will have no depth axis).
- **activation** (`Function`, optional) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or `cntk.initializer`, default `glorot_uniform()`) – initial value of weights *W*
- **pad** (*bool* or tuple of *bool*s, default *False*) – if *False*, then the filter will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, the filter will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **strides** (*int* or tuple of *int*s, default 1) – stride of the convolution (increment when sliding the filter over the input). Use a *tuple* to specify a per-axis value.
- **sharing** (*bool*, default *True*) – weight sharing, must be True for now.
- **bias** (*bool*, optional, default *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer`) – initial value of weights *b*
- **output_shape** (*int* or tuple of *int*s) – output shape. When strides > 2, the output shape is non-deterministic. User can specify the wanted output shape. Note the specified shape must satisify the condition that if a convolution is perform from the output with the same setting, the result must have same shape as the input.
- **reduction_rank** (*int*, default 1) – must be 1 for now. that is stored with tensor shape (H,W) instead of (1,H,W)
- **max_temp_mem_size_in_samples** (*int*, default 0) – set to a positive number to define the maximum workspace memory for convolution.
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: `Function` that accepts one argument and applies the convolution operation to it

---

ConvolutionTranspose1D(*filter_shape, num_filters, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, output_shape=None, name=''*) [source]

Layer factory function to create a 1D convolution transpose layer with optional non-linearity. Same as *ConvolutionTranspose()* except that filter_shape is verified to be 1-dimensional. See *ConvolutionTranspose()* for extensive documentation.

ConvolutionTranspose2D(*filter_shape, num_filters, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, output_shape=None, name=''*) [source]

Layer factory function to create a 2D convolution transpose layer with optional non-linearity. Same as *ConvolutionTranspose()* except that filter_shape is verified to be 2-dimensional. See *ConvolutionTranspose()* for extensive documentation.

ConvolutionTranspose3D(*filter_shape, num_filters, activation=identity, init=glorot_uniform(), pad=False, strides=1, bias=True, init_bias=0, output_shape=None, name=''*) [source]

Layer factory function to create a 3D convolution transpose layer with optional non-linearity. Same as *ConvolutionTranspose()* except that filter_shape is verified to be 3-dimensional. See *ConvolutionTranspose()* for extensive documentation.

Dense(*shape, activation=identity, init=glorot_uniform(), input_rank=None, map_rank=None, bias=True, init_bias=0, name=''*) [source]

Layer factory function to create an instance of a fully-connected linear layer of the form *activation(input @ W + b)* with weights *W* and bias *b*, and *activation* and *b* being optional. *shape* may describe a tensor as well.

A `Dense` layer instance owns its parameter tensors *W* and *b*, and exposes them as attributes `.W` and `.b`.

Example

```
>>> f = Dense(5, activation=C.relu)
>>> x = C.input_variable(3)
>>> h = f(x)
>>> h.shape
    (5,)
>>> f.W.shape
    (3, 5)
>>> f.b.value
    array([ 0.,  0.,  0.,  0.,  0.], dtype=float32)
```

```
>>> # activation through default options
>>> with C.default_options(activation=C.relu):
...     f = Dense(500)
```

The `Dense` layer can be applied to inputs that are tensors, not just vectors. This is useful, e.g., at the top of a image-processing cascade, where after many convolutions with padding and strides it is difficult to know the precise dimensions. For this case, CNTK has an extended definition of matrix product, in which the input tensor will be treated as if it had been automatically flattened. The weight matrix will be a tensor that reflects the "flattened" dimensions in its axes.

Example

```
>>> f = Dense(5, activation=C.softmax) # a 5-class classifier
>>> x = C.input_variable((64,16,16)) # e.g. an image reduced by a convolution stack
>>> y = f(x)
>>> y.shape
(5,)
>>> f.W.shape  # "row" dimension of "matrix" consists of 3 axes that match the input
(64, 16, 16, 5)
```

This behavior can be modified by telling CNTK either the number of axes that should not be projected (`map_rank`) or the rank of the input (`input_rank`). If neither is specified, all input dimensions are projected, as in the example above.

Example

```
>>> f = Dense(5, activation=C.softmax, input_rank=2) # a 5-class classifier
>>> x = C.input_variable((10, 3, 3)) # e.g. 10 parallel 3x3 objects. Input has input_rank=2
axes
>>> y = f(x)
>>> y.shape  # the 10 parallel objects are classified separately, the "10" dimension is
retained
(10, 5)
>>> f.W.shape  # "row" dimension of "matrix" consists of (3,3) matching the input axes to
project
(3, 3, 5)
```

```
>>> f = Dense(5, activation=C.softmax, map_rank=2)
>>> x = C.input_variable((4, 6, 3, 3, 3)) # e.g. 24 parallel 3x3x3 objects arranged in a
4x6 grid. The grid is to be retained
>>> y = f(x)
>>> y.shape  # the 4x6 elements are classified separately, the grid structure is retained
(4, 6, 5)
>>> f.W.shape  # "row" dimension of "matrix" consists of (3,3) matching the input axes to
project
(3, 3, 3, 5)
>>> z = y([np.zeros(x.shape)])
>>> assert z.shape == (1, 4, 6, 5)
```

Parameters:

- **shape** (*int* or *tuple* of *ints*) – vector or tensor dimension of the output of this layer
- **activation** (`Function` , defaults to identity) – optional function to apply at the end, e.g. *relu*
- **init** (scalar or NumPy array or `cntk.initializer` , defaults to `glorot_uniform()`) – initial value of weights *W*
- **input_rank** (int, defaults to *None*) – number of inferred axes to add to W (*map_rank* must not be given)
- **map_rank** (int, defaults to *None*) – expand W to leave exactly *map_rank* axes (*input_rank* must not be given)
- **bias** (bool, optional, defaults to *True*) – the layer will have no bias if *False* is passed here
- **init_bias** (scalar or NumPy array or `cntk.initializer` , defaults to 0) – initial value of weights *b*
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

Dropout(*dropout_rate=None, keep_prob=None, seed=4294967293, name=''*) [source]

Layer factory function to create a drop-out layer.

The dropout rate can be specified as the probability of *dropping* a value (`dropout_rate`). E.g. `Dropout(0.3)` means “drop 30% of the activation values.” Alternatively, it can also be specified as the probability of *keeping* a value (`keep_prob`).

The dropout operation is only applied during training. During testing, this is a no-op. To make sure that this leads to correct results, the dropout operation in training multiplies the result by (1/(1- `dropout_rate`)).

Example

```
>>> f = Dropout(0.2)   # "drop 20% of activations"
>>> h = C.input_variable(3)
>>> hd = f(h)
```

```
>>> f = Dropout(keep_prob=0.8)   # "keep 80%"
>>> h = C.input_variable(3)
>>> hd = f(h)
```

Parameters:

- **dropout_rate** (*float*) – probability of dropping out an element, mutually exclusive with `keep_prob`
- **keep_prob** (*float*) – probability of keeping an element, mutually exclusive with `dropout_rate`
- **seed** (*int*) – random seed.
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

`Embedding`*(shape=None, init=glorot_uniform(), weights=None, name='')* [source]

Layer factory function to create a embedding layer.

An embedding is conceptually a lookup table. For every input token (e.g. a word or any category label), the corresponding entry in the lookup table is returned.

In CNTK, discrete items such as words are represented as one-hot vectors. The table lookup is realized as a matrix product, with a matrix whose rows are the embedding vectors. Note that multiplying a matrix from the left with a one-hot vector is the same as copying out the row for which the input vector is 1. CNTK has special optimizations to make this operation as efficient as an actual table lookup if the input is sparse.

The lookup table in this layer is learnable, unless a user-specified one is supplied through the `weights` parameter. For example, to use an existing embedding table from a file in numpy format, use this:

```
Embedding(weights=np.load('PATH.npy'))
```

To initialize a learnable lookup table with a given numpy array that is to be used as the initial value, pass that array to the `init` parameter (not `weights`).

An `Embedding` instance owns its weight parameter tensor *E*, and exposes it as an attribute `.E`.

Example

```
>>> # learnable embedding
>>> f = Embedding(5)
>>> x = C.input_variable(3)
>>> e = f(x)
>>> e.shape
    (5,)
>>> f.E.shape
    (3, 5)
```

```
>>> # user-supplied embedding
>>> f = Embedding(weights=[[.5, .3, .1, .4, .2], [.7, .6, .3, .2, .9]])
>>> f.E.value
    array([[ 0.5,  0.3,  0.1,  0.4,  0.2],
           [ 0.7,  0.6,  0.3,  0.2,  0.9]], dtype=float32)
>>> x = C.input_variable(2, is_sparse=True)
>>> e = f(x)
>>> e.shape
    (5,)
>>> e(C.Value.one_hot([[1], [0], [0], [1]], num_classes=2))
array([[ 0.7,  0.6,  0.3,  0.2,  0.9],
       [ 0.5,  0.3,  0.1,  0.4,  0.2],
       [ 0.5,  0.3,  0.1,  0.4,  0.2],
       [ 0.7,  0.6,  0.3,  0.2,  0.9]], dtype=float32)
```

Parameters:

- **shape** (*int* or *tuple* of *ints*) – vector or tensor dimension of the output of this layer
- **init** (scalar or NumPy array or `cntk.initializer`, defaults to `glorot_uniform()`) – (learnable embedding only) initial value of weights *E*
- **weights** (NumPy array, mutually exclusive with `init`, defuats to *None*) – (user-supplied embedding only) the lookup table. The matrix rows are the embedding vectors, `weights[i,:]` being the embedding that corresponds to input category *i*.
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the embedding operation to it

Return type: cntk.ops.functions.Function

`GlobalAveragePooling(name='')` [source]

Layer factory function to create a global average-pooling layer.

The global average-pooling operation computes the element-wise mean over all items on an N-dimensional grid, such as an image.

This operation is the same as applying `reduce_mean()` to all grid dimensions.

Example

```
>>> f = GlobalAveragePooling()
>>> f.update_signature((1,4,4))
>>> im = np.array([[[3, 5, 2, 6], [4, 2, 8, 3], [1, 6, 4, 7], [7, 3, 5, 9]]])  # a 4x4
image (feature-map depth 1 for simplicity)
>>> im
    array([[[3, 5, 2, 6],
            [4, 2, 8, 3],
            [1, 6, 4, 7],
            [7, 3, 5, 9]]])
>>> f([[im]])
    array([[[[ 4.6875]]]], dtype=float32)
```

Parameters: **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

GlobalMaxPooling(*name=''*) [source]

Layer factory function to create a global max-pooling layer.

The global max-pooling operation computes the element-wise maximum over all items on an N-dimensional grid, such as an image.

This operation is the same as applying `reduce_max()` to all grid dimensions.

Example

```
>>> f = GlobalMaxPooling()
>>> f.update_signature((1,4,4))
>>> im = np.array([[[3, 5, 2, 6], [4, 2, 8, 3], [1, 6, 4, 7], [7, 3, 5, 9]]])  # a 4x4
image (feature-map depth 1 for simplicity)
>>> im
    array([[[3, 5, 2, 6],
            [4, 2, 8, 3],
            [1, 6, 4, 7],
            [7, 3, 5, 9]]])
>>> f([[im]])
    array([[[[ 9.]]]], dtype=float32)
```

Parameters: **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

Label(*name*) [source]

Layer factory function to create a dummy layer with a given name. This can be used to access an intermediate value flowing through computation.

Parameters: **name** (*str*) – the name of the function instance in the network

Example

```
>>> model = Dense(500) >> Label('hidden') >> Dense(10)
>>> model.update_signature(10)
>>> intermediate_val = model.hidden
>>> intermediate_val.shape
    (500,)
```

Returns: A function that accepts one argument and returns it with the desired name attached

Return type: cntk.ops.functions.Function

LayerNormalization(*initial_scale=1, initial_bias=0, epsilon=0.00001, name=''*) [source]

Layer factory function to create a function that implements layer normalization.

Layer normalization applies this formula to every input element (element-wise): `y = (x - mean(x)) / (stddev(x) + epsilon) * scale + bias` where `scale` and `bias` are learned scalar parameters.

Example

```
>>> f = LayerNormalization(initial_scale=2, initial_bias=1)
>>> f.update_signature(4)
>>> f([np.array([4,0,0,4])])  # result has mean 1 and standard deviation 2, reflecting the
initial values for scale and bias
    array([[ 2.99999, -0.99999, -0.99999,  2.99999]], dtype=float32)
```

Parameters:

- **initial_scale** (*float, default 1*) – initial value for the `scale` parameter
- **initial_bias** (*float, default 0*) – initial value for the `bias` parameter
- **epsilon** (*float, default 0.00001*) – epsilon added to the standard deviation to avoid division by 0
- **name** (*str, optional*) – the name of the Function instance in the network

Returns: A function that accepts one argument and applies the operation to it

Return type: cntk.ops.functions.Function

MaxPooling(*filter_shape, strides=1, pad=False, name=''*) [source]

Layer factory function to create a max-pooling layer.

Like `Convolution()`, `MaxPooling()` processes items arranged on an N-dimensional grid, such as an image. Typically, each item is a vector. For each item, max-pooling computes the element-wise maximum over a window ("receptive field") of items surrounding the item's position on the grid.

The size (spatial extent) of the receptive field is given by `filter_shape`. E.g. for 2D pooling, `filter_shape` should be a tuple of two integers, such as *(5,5)*.

Example

```
>>> f = MaxPooling((3,3), strides=2)  # reduce dimensionality by 2, pooling over windows of
3x3
>>> h = C.input_variable((32,240,320))  # e.g. 32-dim feature map
>>> hp = f(h)
>>> hp.shape  # spatial dimension has been halved due to stride, and lost one due to 3x3
window without padding
    (32, 119, 159)
```

```
>>> f = MaxPooling((2,2), strides=2)
>>> f.update_signature((1,4,4))
>>> im = np.array([[[3, 5, 2, 6], [4, 2, 8, 3], [1, 6, 4, 7], [7, 3, 5, 9]]])  # a 4x4
image (feature-map depth 1 for simplicity)
>>> im
    array([[[3, 5, 2, 6],
            [4, 2, 8, 3],
            [1, 6, 4, 7],
            [7, 3, 5, 9]]])
>>> f([[im]])  # due to strides=2, this picks the max out of each 2x2 sub-block
    array([[[[ 5.,  8.],
             [ 7.,  9.]]]], dtype=float32)
```

Parameters:

- **filter_shape** (*int* or *tuple* of *ints*) – shape (spatial extent) of the receptive field, *not* including the input feature-map depth. E.g. (3,3) for a 2D convolution.
- **strides** (*int* or *tuple* of *ints*, defaults to 1) – stride (increment when sliding over the input). Use a *tuple* to specify a per-axis value.
- **pad** (*bool* or *tuple* of *bools*, defaults to *False*) – if *False*, then the pooling operation will be shifted over the "valid" area of input, that is, no value outside the area is used. If `pad=True` on the other hand, pooling will be applied to all input positions, and positions outside the valid region will be considered containing zero. Use a *tuple* to specify a per-axis value.
- **name** (*str, defaults to ''*) – the name of the function instance in the network

Returns: A function that accepts one argument and applies the max-pooling operation to it

Return type: cntk.ops.functions.Function

MaxUnpooling(*filter_shape, strides=1, pad=False, name=''*) [source]