Layers Library Reference

Note: This documentation has not yet been completely updated with respect to the latest update of the Layers library. It should be correct but misses several new options and layer types.

CNTK predefines a number of common “layers,” which makes it very easy to write simple networks that consist of standard layers layered on top of each other. Layers are function objects that can be used like a regular `Function` but hold learnable parameters and have an additional pair of `()` to pass construction parameters or attributes.

For example, this is the network description for a simple 1-hidden layer model using the `Dense()` layer:

```
h = Dense(1024, activation=relu)(features)
p = Dense(9000, activation=softmax)(h)
```

which can then, e.g., be used for training against a cross-entropy criterion:

```
ce = cross_entropy(p, labels)
```

If your network is a straight concatenation of operations (many are), you can use the alternative Sequential() notation:

```
from cntk.layers import *
my_model = Sequential ([
    Dense(1024, activation=relu),
    Dense(9000, activation=softmax)
])
```

and invoke it like this:

```
p = my_model(features)
```

Built on top of `Sequential()` is For(), which allows to easily create models with repetitions. For example, a 2011-style feed-forward speech-recognition network with 6 hidden sigmoid layers of identical dimensions can be written like this:

```
my_model = Sequential ([
    For(range(6), lambda: \
        Dense(2048, activation=sigmoid))
    Dense(9000, activation=softmax)
])
```

Note that for most real-life inference scenarios, the output layer's `softmax` non-linearity is not needed (it is instead made part of the training criterion).

General patterns

Specifying the same options to multiple layers

Often, many layers share options. For example, typical image-recognition systems use the `relu` activation function throughout. You can use the Python `with` statement with the CNTK `default_options()` function to define scopes with locally changed defaults, using one of the following two forms:

```
with default_options(OPT1=VAL1, OPT2=VAL2, ...):
    # scope with modified defaults

with default_options_for(FUNCTION, OPT1=VAL1, OPT2=VAL2, ...):
    # scope with modified defaults for FUNCTION only
```

The following options can be overridden with the `with` statement:

- `init` (default: `glorot_uniform()`): initializer specification, for Dense(), Convolution(), and Embedding()
- `activation` (default: `None`): activation function, for `Dense()` and `Convolution()`
- `bias` (default: `True`): have a bias, for `Dense()` and `Convolution()`
- `init_bias` (default: `0`): initializer specification for the bias, for `Dense()` and `Convolution()`
- `initial_state` (default: `None`): initial state to use in `Recurrence()` Recurrence()
- `use_peepholes` (default: `False`): use peephole connections in `LSTM()` LSTM(), GRU(), RNNStep()

The second form allows to set default options on a per-layer type. This is, for example, valuable for the `pad` parameter which enables padding in convolution and pooling, but is not always set to the same for these two layer types.

Weight sharing

If you assign a layer to a variable and use it in multiple places, *the weight parameters will be shared*. If you say

```
lay = Dense(1024, activation=sigmoid)
h1 = lay(x)
h2 = lay(h1)  # same weights as h1
```

`h1` and `h2` will *share the same weight parameters*, as `lay()` is the *same function* in both cases. In the above case this is probably not what was desired, so be aware. If both invocations of `lay()` above are meant to have different parameters, remember to define two separate instances, for example `lay1 = Dense(...)` and `lay2 = Dense(...)`.

So why this behavior? Layers allow to share parameters across sections of a model. Consider a DSSM model which processes two input images, say `doc` and `query` identically with the same processing chain, and compares the resulting hidden vectors:

```
with default_options(activation=relu):
    image_to_vec = Sequential([
        Convolution((5,5), 32, pad=True), MaxPooling((3,3), strides=2),
        Convolution((5,5), 64, pad=True), MaxPooling((3,3), strides=2),
        Dense(64),
        Dense(10, activation=None)
    ])
z_doc   = image_to_vec (doc)
z_query = image_to_vec (query)  # same model as for z_doc
sim = cos_distance(zdoc, z_query)
```

where `image_to_vec` is the part of the model that converts images into flat vector. `image_to_vec` is a function object that in turn contains several function objects (e.g. three instances of `Convolution()`). `image_to_vec` is instantiated *once*, and this instance holds the learnable parameters of all the included function objects. Both invocations of `model()` will share these parameters in application, and their gradients will be the sum of both invocations.

Lastly, note that if in the above example `query` and `doc` must have the same dimensions, since they are processed through the same function object, and that function object's first layer has its input dimension inferred to match that of both `query` and `doc`. If their dimensions differ, then this network is malformed, and dimension inference/validation will fail with an error message.

Example models

The following shows a slot tagger that embeds a word sequence, processes it with a recurrent LSTM, and then classifies each word:

```
from cntk.layers import *
tagging_model = Sequential ([
    Embedding(150),         # embed into a 150-dimensional vector
    Recurrence(LSTM(300)),  # forward LSTM
    Dense(labelDim)         # word-wise classification
])
```

And the following is a simple convolutional network for image recognition, using the `with default_options(...):` Specifying the same options to multiple layers pattern):

```
with default_options(activation=relu):
    conv_net = Sequential ([
        # 3 layers of convolution and dimension reduction by pooling
        Convolution((5,5), 32, pad=True), MaxPooling((3,3), strides=2),
        Convolution((5,5), 32, pad=True), MaxPooling((3,3), strides=2),
        Convolution((5,5), 64, pad=True), MaxPooling((3,3), strides=2),
        # 2 dense layers for classification
        Dense(64),
        Dense(10, activation=None)
    ])
```

Notes

Many layers are wrappers around underlying CNTK primitives, along with the respective required learnable parameters. For example, `` `Convolution() `` Convolution() wraps the `convolution()` primitive. The benefits of using layers are: * layers contain learnable parameters of the correct dimension * layers are composable (cf. `` `Sequential() `` Sequential())

However, since the layers themselves are implemented in Python using the same CNTK primitives that are available to the user, if you find that a layer you need is not available, you can always write it yourself or write the formula directly as a CNTK expression.

The Python library described here is the equivalent of BrainScript's Layers Library.

Dense()

Factory function to create a fully-connected layer. `Dense()` takes an optional activation function.

```
Dense(shape, activation=default_override_or(identity),
init=default_override_or(glorot_uniform()),
      input_rank=None, map_rank=None,
      bias=default_override_or(True), init_bias=default_override_or(0),
      name='')
```

Parameters

- `shape`: output dimension of this layer
- `activation` (default: `None`: pass a function here to be used as the activation function, such as `activation=relu`
- `input_rank`: if given, number of trailing dimensions that are transformed by `Dense()` (`map_rank` must not be given)
- `map_rank`: if given, the number of leading dimensions that are not transformed by `Dense()` (`input_rank` must not be given)
- `init` (default: `glorot_uniform()`): initializer descriptor for the weights. See `cntk.initializer` for a full list of random-initialization options.
- `bias`: if `False`, do not include a bias parameter
- `init_bias` (default: `0`): initializer for the bias

Return Value

A function that implements the desired fully-connected layer. See description.

Description

Use these factory functions to create a fully-connected layer. It creates a function object that contains a learnable weight matrix and, unless `bias=False`, a learnable bias. The function object can be used like a function, which implements one of these formulas (using Python 3.5 `@` operator for matrix multiplication):

```
Dense(...)(v) = activation (v @ W + b)
Dense(...)(v) = v @ W + b      # if activation is None
```

where `W` is a weight matrix of dimension `((dimension of v), shape)`, `b` is the bias of dimension `(outdim,)`, and the resulting value has dimension (or tensor dimensions) as given by `shape`.

Tensor support

If the returned function is applied to an input of a tensor rank > 1, e.g. a 2D image, `W` will have the dimension `(..., (second dimension of input), (first dimension of input), shape)`.

On the other hand, `shape` can be a vector that specifies tensor dimensions, for example `(10,10)`. In that case, `W` will have the dimension `((dimension of input), ..., shape[1], shape[0])`, and `b` will have the tensor dimensions `(..., shape[1], shape[0])`.

CNTK's matrix product will interpret these extra output or input dimensions as if they were flattened into a long vector. For more details on this, see the documentation of Times().

The options `input_rank` and `map_rank`, which are mutually exclusive, can specify that not all of the input axes of a tensor should be transformed. `map_rank` specifies how many leading axes are kept as dimensions in the result; those axes are not part of the projection, but rather each element along these axes is transformed independently (aka *mapped*). `input_rank` is an alternative that instead specifies the how many trailing axes are to be transformed (the remaining are mapped).

Example:

```
h = Dense(1024, activation=sigmoid)(v)
```

or alternatively:

```
layer = Dense(1024, activation=sigmoid)
h = layer(v)
```

Convolution()

Creates a convolution layer with optional non-linearity.

```
Convolution(filter_shape,     # shape of receptive field, e.g. (3,3)
            num_filters=None, # e.g. 64 or None (which means 1 channel and don't add a
dimension)
            sequential=False, # time convolution if True (filter_shape[0] corresponds to
dynamic axis)
            activation=default_override_or(identity),
            init=default_override_or(glorot_uniform()),
            pad=default_override_or(False),
            strides=1,
            bias=default_override_or(True),
            init_bias=default_override_or(0),
            reduction_rank=1, # (0 means input has no depth dimension, e.g. audio signal or B&W
image)
            max_temp_mem_size_in_samples=0,
            name='')
```

Parameters

- `filter_shape`: shape of receptive field of the filter, e.g. `(5,5)` for a 2D filter (not including the input feature-map depth)
- `num_filters`: number of output channels (number of filters)
- `activation`: optional non-linearity, e.g. `activation=relu`
- `init`: initializer descriptor for the weights, e.g. `glorot_uniform()`. See `cntk.initializer` for a full list of random-initialization options.
- `pad`: if `False` (default), then the filter will be shifted over the “valid” area of input, that is, no value outside the area is used. If `pad` is `True` on the other hand, the filter will be

applied to all input positions, and values outside the valid region will be considered zero.

- `strides`: increment when sliding the filter over the input. E.g. `(2,2)` to reduce the dimensions by 2
- `bias`: if `False`, do not include a bias parameter
- `init_bias`: initializer for the bias
- `use_correlation`: currently always `True` and cannot be changed. It indicates that `Convolution()` actually computes the cross-correlation rather than the true convolution

Return Value

A function that implements the desired convolution operation.

Description

Use these factory functions to create a convolution layer.

The resulting layer applies a convolution operation on N-dimensional feature maps. The caller specifies the receptive field of the filter and the number of filters (output feature maps).

A set of filters for a given receptive field (e.g. `(5,5)`) is correlated with every location of the input (e.g. a `(480, 640)`-sized image). Assuming padding is enabled (`pad`) and strides are 1, this will generate an output of the same dimension (`(480, 640)`).

Typically, many filters are applied at the same time, to create "per-pixel activation vectors". `num_filters` specifies the number: For every input location, an entire vector of `num_filters` is produced. For our example above, setting `num_filters` to 64 would in a `(64, 480, 640)`-sized tensor. That first axis is also called the *channel dimension* or the *feature-map axis*.

When convolution is applied to an input with a channel dimension, each filter will also consist of vectors of the input's channel dimension. E.g. when applying convolution with a specified receptive field of `(5,5)` to a `(3, 480, 640)`-sized color image, each filter will be a `(3, 5, 5)]` tensor.

All `num_filters` filters are stacked together into the so-called convolution *kernel*, which is a parameter tensor owned by and held inside this layer. In our example, the kernel shape will be `(64, 3, 5, 5)`.

The following summarizes the relationship between the various dimensions and shapes:

```
input shape   : (               num_input_channels, (spatial dims) )
filter shape  : (                                   (filter_shape) )
output shape  : ( num_filters,                      (spatial dims) )
kernel shape  : ( num_filters,  num_input_channels, (filter_shape)     )
```

which in our example are:

```
input shape   : (
```