DyNN Documentation

Release 0.0.12

Paul Michel

Sep 21, 2018

Contents:

CHAPTER 1

DyNN

1.1 dynn package

1.1.1 DyNN

1.1.2 Subpackages

1.1.2.1 dynn.data package

Data

This module contains helper functions and classes to manage data. This includes code for minibatching as well as functions for downloading common datasets.

Supported datasets are:

- MNIST
- CIFAR-10

Submodules

Batching procedures

Iterators implementing common batching strategies.

class dynn.data.batching.**NumpyBatchIterator**(*data*, *targets*, *batch_size=32*, *shuffle=True*)

Bases: object

Wraps a list of numpy arrays and a list of targets as a batch iterator.

You can then iterate over this object and get tuples of `batch_data, batch_targets` ready for use in your computation graph.

Example for classification:

```
# 1000 10-dimensional inputs
data = np.random.uniform(size=(1000, 10))
# Class labels
labels = np.random.randint(10, size=1000)
# Iterator
batched_dataset = NumpyBatchIterator(data, labels, batch_size=20)
# Training loop
for x, y in batched_dataset:
    # x has shape (10, 20) while y has shape (20,)
    # Do something with x and y
```

Example for multidimensional regression:

```
# 1000 10-dimensional inputs
data = np.random.uniform(size=(1000, 10))
# 5-dimensional outputs
labels = np.random.uniform(size=(1000, 5))
# Iterator
batched_dataset = NumpyBatchIterator(data, labels, batch_size=20)
# Training loop
for x, y in batched_dataset:
    # x has shape (10, 20) while y has shape (5, 20)
    # Do something with x and y
```

Parameters

- **data** (*list*) – List of numpy arrays containing the data
- **targets** (*list*) – List of targets
- **batch_size** (*int, optional*) – Batch size (default: 32)
- **shuffle** (*bool, optional*) – Shuffle the dataset whenever starting a new iteration (default: True)

__getitem__(*index*)

Returns the index th **batch** (not sample)

This returns something different every time the data is shuffled.

The result is a tuple batch_data, batch_target where each of those is a numpy array in Fortran layout (for more efficient input in dynet). The batch size is always the last dimension.

Parameters **index** (*int, slice*) – Index or slice

Returns batch_data, batch_target

Return type tuple

__init__(*data*, *targets*, *batch_size=32*, *shuffle=True*)

Initialize self. See help(type(self)) for accurate signature.

__len__()

This returns the number of **batches** in the dataset (not the total number of samples)

Returns

Number of batches in the dataset ceil(len(data)/batch_size)

Return type int

__weakref__
list of weak references to the object (if defined)

just_passed_multiple(*batch_number*)
Checks whether the current number of batches processed has just passed a multiple of `batch_number`.

For example you can use this to report at regular interval (eg. every 10 batches)

Parameters **batch_number** (*int*) – [description]

Returns `True` if $_batch$

Return type bool

percentage_done()
What percent of the data has been covered in the current epoch

reset()
Reset the iterator and shuffle the dataset if applicable

CIFAR10

Various functions for accessing the CIFAR10 dataset.

dynn.data.cifar10.**download_cifar10**(*path='.'*, *force=False*)
Downloads CIFAR10 from "https://www.cs.toronto.edu/~kriz/cifar.html"

Parameters

- **path** (*str, optional*) – Local folder (defaults to ".")
- **force** (*bool, optional*) – Force the redownload even if the files are already at `path`

dynn.data.cifar10.**load_cifar10**(*path*)
Loads the CIFAR10 dataset

Returns the train and test set, each as a list of images and a list of labels. The images are represented as numpy arrays and the labels as integers.

Parameters **path** (*str*) – Path to the folder containing the `*-ubyte.gz` files

Returns train and test sets

Return type tuple

dynn.data.cifar10.**read_cifar10**(*split*, *path*)
Iterates over the CIFAR10 dataset

Example:

```
for image in read_cifar10("train", "/path/to/cifar10"):
    train(image)
```

Parameters

- **split** (*str*) – Either `"training"` or `"test"`
- **path** (*str*) – Path to the folder containing the `*-ubyte` files

Returns image, label

Return type tuple

Data utilities

Helper functions to download and manage datasets.

dynn.data.data_util.**download_if_not_there**(*file*, *url*, *path*, *force=False*)
Downloads a file from the given url if and only if the file doesn't already exist in the provided path or `force=True`

Parameters

- **file** (*str*) – File name
- **url** (*str*) – Url where the file can be found (without the filename)
- **path** (*str*) – Path to the local folder where the file should be stored
- **force** (*bool, optional*) – Force the file download (useful if you suspect that the file might have changed)

MNIST

Various functions for accessing the MNIST dataset.

dynn.data.mnist.**download_mnist**(*path='.'*, *force=False*)
Downloads MNIST from "http://yann.lecun.com/exdb/mnist/"

Parameters

- **path** (*str, optional*) – Local folder (defaults to ".")
- **force** (*bool, optional*) – Force the redownload even if the files are already at `path`

dynn.data.mnist.**load_mnist**(*path*)
Loads the MNIST dataset

Returns the train and test set, each as a list of images and a list of labels. The images are represented as numpy arrays and the labels as integers.

Parameters **path** (*str*) – Path to the folder containing the `*-ubyte.gz` files

Returns train and test sets

Return type tuple

dynn.data.mnist.**read_mnist**(*split*, *path*)
Iterates over the MNIST dataset

Example:

```
for image in read_mnist("train", "/path/to/mnist"):
    train(image)
```

Parameters

- **split** (*str*) – Either `"training"` or `"test"`
- **path** (*str*) – Path to the folder containing the `*-ubyte` files

Returns image, label

Return type tuple

1.1.2.2 dynn.layers package

Layers

Layers are the standard unit of neural models in DyNN. Layers are typically used like this:

```
# Instantiate layer
layer = Layer(parameter_collection, *args, **kwargs)
# [...]
# Renew computation graph
dy.renew_cg()
# Initialize layer
layer.init(*args, **kwargs)
# Apply layer forward pass
y = layer(x)
```

Submodules

Base layer

class dynn.layers.base_layers.**BaseLayer**(*name*)
Bases: object

Base layer interface

__call__(**args*, ***kwargs*)
Execute forward pass

__init__(*name*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

init(**args*, ***kwargs*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

class dynn.layers.base_layers.**ParametrizedLayer**(*pc*, *name*)
Bases: *dynn.layers.base_layers.BaseLayer*

This is the base class for layers with trainable parameters

__init__(*pc*, *name*)
Creates a subcollection for this layer with a custom name

init(**args*, ***kwargs*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc... This needs to be implemented for parametrized layers

Combination layers

Perhaps unsurprisingly, combination layers are layers that combine other layers within one layer.

class dynn.layers.combination_layers.**ConcatenatedLayers**(**layers*, *dim=0*, *default_insert_dim=False*)

Bases: *dynn.layers.base_layers.BaseLayer*

A helper class to run layers on the same input and concatenate their outputs

This can be used to create 2d conv layers with multiple kernel sizes by concatenating multiple `dynn.layers.Conv2DLayer`.

Parameters

- **layers** (*list*) – A list of `dynn.layers.BaseLayer` objects. The first layer is the first one applied to the input. It is the programmer's responsibility to make sure that the layers are compatible (eg. each layer takes the same input and the outputs have the same shape everywhere except along the concatenation dimension)
- **dim** (*int*) – The concatenation dimension
- **default_insert_dim** (*bool, optional*) – Instead of concatenating along an existing dimension, insert a a new dimension at `dim` and concatenate.

__call__(*x*, *insert_dim=None*)

Calls all the layers in succession.

Computes `dy.concatenate([layers[0](x) ...layers[n-1](x)], d=dim)`

Parameters

- **x** (`dynet.Expression`) – Input expression
- **default_insert_dim** (*bool, optional*) – Override the default

Returns

Depending on `return_last_only`, returns either the last expression or a list of all the layer's outputs (first to last)

Return type `dynet.Expression`, list

__init__(**layers*, *dim=0*, *default_insert_dim=False*)

Initialize self. See help(type(self)) for accurate signature.

init(*test=False*, *update=True*)

Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

class dynn.layers.combination_layers.**StackedLayers**(**layers*, *default_return_last_only=True*)

Bases: *dynn.layers.base_layers.BaseLayer*

A helper class to stack layers into deep networks.

Parameters

- **layers** (*list*) – A list of `dynn.layers.BaseLayer` objects. The first layer is the first one applied to the input. It is the programmer's responsibility to make sure that the layers are compatible (eg. the output of each layer can be fed into the next one)
- **default_return_last_only** (*bool, optional*) – Return only the output
- **the last layer** (*of*) –

__call__(*x*, *return_last_only=None*)

Calls all the layers in succession.

Computes `layers[n-1](layers[n-2](...layers[0](x)))`

Parameters

- **x** (dynet.Expression) – Input expression
- **return_last_only** (*bool, optional*) – Overrides the default

Returns

Depending on return_last_only, returns either the last expression or a list of all the layer's outputs (first to last)

Return type dynet.Expression, list

__init__ (**layers*, *default_return_last_only=True*)
Initialize self. See help(type(self)) for accurate signature.

init (*test=False*, *update=True*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

Convolution layers

class dynn.layers.convolution_layers.**Conv1DLayer**(*pc*, *input_dim*, *num_kernels*, *kernel_width*, *activation=<function identity>*, *dropout_rate=0.0*, *nobias=False*, *default_zero_padded=True*, *default_stride=1*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*

1D convolution along the first dimension

Parameters

- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters
- **input_dim** (*int*) – Input dimension
- **num_kernels** (*int*) – Number of kernels (essentially the output dimension)
- **kernel_width** (*int*) – Width of the kernels
- **activation** (*function, optional*) – activation function (default: identity)
- **dropout** (*float, optional*) – Dropout rate (default 0)
- **nobias** (*bool, optional*) – Omit the bias (default False)
- **default_zero_padded** (*bool, optional*) – Default padding behaviour. Pad the input with zeros so that the output has the same length (default True)
- **default_stride** (*list, optional*) – Default stride along the length (defaults to 1).

__call__ (*x*, *stride=None*, *zero_padded=None*)
Forward pass

Parameters

- **x** (dynet.Expression) – Input expression with the shape (length, input_dim)
- **stride** (*int, optional*) – Stride along the temporal dimension

- **zero_padded** (*bool, optional*) – Pad the image with zeros so that the output has the same length (default `True`)

Returns Convolved sequence.

Return type `dynet.Expression`

__init__(*pc*, *input_dim*, *num_kernels*, *kernel_width*, *activation=<function identity>*, *dropout_rate=0.0*, *nobias=False*, *default_zero_padded=True*, *default_stride=1*)
Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)
Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to `True`, dropout is not applied (default: `True`)
- **update** (*bool, optional*) – Whether to update the parameters (default: `True`)

class dynn.layers.convolution_layers.**Conv2DLayer**(*pc*, *num_channels*, *num_kernels*, *kernel_size*, *activation=<function identity>*, *dropout_rate=0.0*, *nobias=False*, *default_zero_padded=True*, *default_strides=None*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*

2D convolution

Parameters

- **pc** (`dynet.ParameterCollection`) – Parameter collection to hold the parameters
- **num_channels** (*int*) – Number of channels in the input image
- **num_kernels** (*int*) – Number of kernels (essentially the output dimension)
- **kernel_size** (*list, optional*) – Default kernel size. This is a list of two elements, one per dimension.
- **activation** (*function, optional*) – activation function (default: `identity`)
- **dropout** (*float, optional*) – Dropout rate (default 0)
- **nobias** (*bool, optional*) – Omit the bias (default `False`)
- **default_zero_padded** (*bool, optional*) – Default padding behaviour. Pad the image with zeros so that the output has the same width/height (default `True`)
- **default_strides** (*list, optional*) – Default stride along each dimension (list of size 2, defaults to `[1, 1]`).

__call__(*x*, *strides=None*, *zero_padded=None*)
Forward pass

Parameters

- **x** (`dynet.Expression`) – Input image (3-d tensor) or matrix.
- **zero_padded** (*bool, optional*) – Pad the image with zeros so that the output has the same width/height. If this is not specified, the default specified in the constructor is used.

- **strides** (*list, optional*) – Stride along width/height. If this is not specified, the default specified in the constructor is used.

Returns Convolved image.

Return type dynet.Expression

__init__(*pc*, *num_channels*, *num_kernels*, *kernel_size*, *activation=<function identity>*, *dropout_rate=0.0*, *nobias=False*, *default_zero_padded=True*, *default_strides=None*)
Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)
Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to True, dropout is not applied (default: True)
- **update** (*bool, optional*) – Whether to update the parameters (default: True)

Densely connected layers

class dynn.layers.dense_layers.**DenseLayer**(*pc*, *input_dim*, *output_dim*, *activation=<built-in function tanh>*, *dropout=0.0*, *nobias=False*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*

Densely connected layer

$y = f(Wx + b)$

Parameters

- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters
- **input_dim** (*int*) – Input dimension
- **output_dim** (*int*) – Output dimension
- **activation** (*function, optional*) – activation function (default: dynet.tanh)
- **dropout** (*float, optional*) – Dropout rate (default 0)
- **nobias** (*bool, optional*) – Omit the bias (default False)

__call__(*x*)
Forward pass

Parameters **x** (dynet.Expression) – Input expression (a vector)

Returns $y = f(Wx + b)$

Return type dynet.Expression

__init__(*pc*, *input_dim*, *output_dim*, *activation=<built-in function tanh>*, *dropout=0.0*, *nobias=False*)
Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)
Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to True, dropout is not applied (default: True)

- **update** (*bool, optional*) – Whether to update the parameters (default: `True`)

class dynn.layers.dense_layers.**GatedLayer**(*pc*, *input_dim*, *output_dim*, *activation=<built-in function tanh>*, *dropout=0.0*)

Bases: *dynn.layers.base_layers.ParametrizedLayer*

Gated linear layer:

$y = (W_o x + b_o) \circ \sigma(W_g x + b_g)$

Parameters

- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters
- **input_dim** (*int*) – Input dimension
- **output_dim** (*int*) – Output dimension
- **activation** (*function, optional*) – activation function (default: `dynet.tanh`)
- **dropout** (*float, optional*) – Dropout rate (default 0)

__call__(*x*)

Forward pass

Parameters **x** (dynet.Expression) – Input expression (a vector)

Returns $y = (W_o x + b_o) \circ \sigma(W_g x + b_g)$

Return type dynet.Expression

__init__(*pc*, *input_dim*, *output_dim*, *activation=<built-in function tanh>*, *dropout=0.0*)

Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)

Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to `True`, dropout is not applied (default: `True`)
- **update** (*bool, optional*) – Whether to update the parameters (default: `True`)

Flow related layers

Those layers don't perform any computation in the forward pass.

class dynn.layers.flow_layers.**FlattenLayer**

Bases: *dynn.layers.base_layers.BaseLayer*

Flattens the output such that there is only one dimension left (batch dimension notwithstanding)

Example:

```
# Create the layer
flatten = FlattenLayer()
# Dummy batched 2d input
x = dy.zeros((3, 4), batch_size=7)
# x.dim() -> (3, 4), 7
y = flatten(x)
# y.dim() -> (12,), 7
```

__call__(*x*)
Flattens the output such that there is only one dimension left (batch dimension notwithstanding)

Parameters **x** (*[type]*) – [description]

Returns [description]

Return type [type]

__init__()
Initialize self. See help(type(self)) for accurate signature.

Functional layers

class dynn.layers.functional_layers.**AdditionLayer**(*layer1*, *layer2*)
Bases: *dynn.layers.functional_layers.BinaryOpLayer*

Addition of two layers.

This is the layer returned by the addition syntax:

```
AdditionLayer(layer1, layer2)(x) == layer1(x) + layer2(x)
# is the same thing as
add_1_2 = layer1 + layer2
add_1_2(x) == layer1(x) + layer2(x)
```

Parameters

- **layer1** (base_layers.BaseLayer) – First layer
- **layer2** (base_layers.BaseLayer) – Second layer

__init__(*layer1*, *layer2*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**BinaryOpLayer**(*layer1*, *layer2*, *binary_operation*)
Bases: *dynn.layers.base_layers.BaseLayer*

This layer wraps two layers with a binary operation.

```
BinaryOpLayer(layer1, layer2, op)(x) == op(layer1(x), layer2(x))
```

This is useful to express the addition of two layers as another layer.

Parameters

- **layer1** (base_layers.BaseLayer) – First layer
- **layer2** (base_layers.BaseLayer) – Second layer
- **binary_operation** (*function*) – A binary operation on dynet.Expression objects

__call__(**args*, ***kwargs*)
Execute forward pass

__init__(*layer1*, *layer2*, *binary_operation*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

class dynn.layers.functional_layers.**CmultLayer**(*layer1*, *layer2*)
Bases: *dynn.layers.functional_layers.BinaryOpLayer*

Coordinate-wise multiplication of two layers.

```
CmultLayer(layer1, layer2)(x) == dy.cmult(layer1(x), layer2(x))
```

Parameters

- **layer1** (base_layers.BaseLayer) – First layer
- **layer2** (base_layers.BaseLayer) – Second layer

__init__(*layer1*, *layer2*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**ConstantLayer**(*constant*)
Bases: *dynn.layers.base_layers.BaseLayer*

This is the "zero"-ary layer.

```
# Takes in numbers
ConstantLayer(5)() == dy.inputTensor([5])
# Or lists
ConstantLayer([5, 6])() == dy.inputTensor([5, 6])
# Or numpy arrays
ConstantLayer(np.ones((10, 12)))() == dy.inputTensor(np.ones((10, 12)))
```

Parameters **constant** (*number, np.ndarray*) – The constant. It must be a type that can be turned into a dynet.Expression

__call__(**args*, ***kwargs*)
Execute forward pass

__init__(*constant*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

class dynn.layers.functional_layers.**IdentityLayer**
Bases: *dynn.layers.functional_layers.LambdaLayer*

The identity layer does literally nothing

```
IdentityLayer()(x) == x
```

It passes its input directly as the output. Still, it can be useful to express more complicated layers like residual connections.

__init__()
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**LambdaLayer**(*function*)
Bases: *dynn.layers.base_layers.BaseLayer*

This layer applies an arbitrary function to its input.

```
LambdaLayer(f)(x) == f(x)
```

This is useful if you want to wrap activation functions as layers. The unary operation should be a function taking dynet.Expression to dynet.Expression.

You shouldn't use this to stack layers though, op oughtn't be a layer. If you want to stack layers, use combination_layers.StackedLayers.

Parameters

- **layer** (base_layers.BaseLayer) – The layer to which output you want to apply the unary operation.
- **binary_operation** (*function*) – A unary operation on dynet.Expression objects

__call__(**args*, ***kwargs*)
Returns function(*args, **kwargs)

__init__(*function*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**NegationLayer**(*layer*)
Bases: *dynn.layers.functional_layers.UnaryOpLayer*

Negates the output of another layer:

```
NegationLayer(layer)(x) == - layer(x)
```

It can also be used with the - syntax directly:

```
negated_layer = - layer
# is the same as
negated_layer = NegationLayer(layer)
```

Parameters **layer** (base_layers.BaseLayer) – The layer to which output you want to apply the negation.

__init__(*layer*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**SubstractionLayer**(*layer1*, *layer2*)
Bases: *dynn.layers.functional_layers.BinaryOpLayer*

Substraction of two layers.

This is the layer returned by the substraction syntax:

```
SubstractionLayer(layer1, layer2)(x) == layer1(x) - layer2(x)
# is the same thing as
add_1_2 = layer1 - layer2
add_1_2(x) == layer1(x) - layer2(x)
```

Parameters

- **layer1** (base_layers.BaseLayer) – First layer

- **layer2** (base_layers.BaseLayer) – Second layer

__init__(*layer1*, *layer2*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.functional_layers.**UnaryOpLayer**(*layer*, *unary_operation*)
Bases: *dynn.layers.base_layers.BaseLayer*

This layer wraps a unary operation on another layer.

```
UnaryOpLayer(layer, op)(x) == op(layer(x))
```

This is a shorter way of writing:

```
UnaryOpLayer(layer, op)(x) == StackedLayers(layer, LambdaLayer(op))
```

You shouldn't use this to stack layers though, op oughtn't be a layer. If you want to stack layers, use combination_layers.StackedLayers.

Parameters

- **layer** (base_layers.BaseLayer) – The layer to which output you want to apply the unary operation.
- **binary_operation** (*function*) – A unary operation on dynet.Expression objects

__call__(**args*, ***kwargs*)
Returns unary_operation(layer(*args, **kwargs))

__init__(*layer*, *unary_operation*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Initialize the wrapped layer

Normalization layers

class dynn.layers.normalization_layers.**LayerNormalization**(*input_dim*, *pc*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*

Layer normalization layer:

$y = \frac{g}{\sigma(x)} \cdot (x - \mu(x) + b)$

Parameters

- **input_dim** (*int*) – Input dimension
- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters

__call__(*x*)
Layer-normalize the input

Parameters **x** (dynet.Expression) – Input expression

Returns $y = \frac{g}{\sigma(x)} \cdot (x - \mu(x) + b)$

Return type dynet.Expression

__init__(*input_dim*, *pc*)
Creates a subcollection for this layer with a custom name

init(*update=True*)
Initialize the layer before performing computation

Parameters **update** (*bool, optional*) – Whether to update the parameters (default: `True`)

Pooling layers

class dynn.layers.pooling_layers.**MaxPooling1DLayer**(*default_kernel_size=None*, *default_stride=1*)
Bases: *dynn.layers.base_layers.BaseLayer*

1D max pooling

Parameters

- **default_kernel_size** (*int, optional*) – Default kernel size. If this is not specified, the default is to pool over the full sequence (default: `None`)
- **stride** (*int, optional*) – Default temporal stride (default: `1`)

__call__(*x*, *kernel_size=None*, *stride=None*)
Max pooling over the first dimension.

This takes either a list of `N` `d`-dimensional vectors or a `N x d` matrix.

The output will be a matrix of dimension `(N - kernel_size + 1) // stride x d`

Parameters

- **x** (`dynet.Expression`) – Input matrix or list of vectors
- **dim** (*int, optional*) – The reduction dimension (default: `0`)
- **kernel_size** (*int, optional*) – Kernel size. If this is not specified, the default size specified in the constructor is used.
- **stride** (*int, optional*) – Temporal stride. If this is not specified, the default stride specified in the constructor is used.

Returns Pooled sequence.

Return type `dynet.Expression`

__init__(*default_kernel_size=None*, *default_stride=1*)
Initialize self. See help(type(self)) for accurate signature.

class dynn.layers.pooling_layers.**MaxPooling2DLayer**(*default_kernel_size=None*, *default_strides=None*)
Bases: *dynn.layers.base_layers.BaseLayer*

2D max pooling.

Parameters

- **kernel_size** (*list, optional*) – Default kernel size. This is a list of two elements, one per dimension. If either is not specified, the default is to pool over the entire dimension (default: `[None, None]`)
- **default_strides** (*list, optional*) – Stride along each dimension (list of size 2, defaults to `[1, 1]`).

__call__(*x*, *kernel_size=None*, *strides=None*)
Max pooling over the first dimension.

If either of the `kernel_size` elements is not specified, the pooling will be done over the full dimension (and the stride is ignored)

Parameters

- **x** (dynet.Expression) – Input image (3-d tensor) or matrix.
- **kernel_size** (*list, optional*) – Size of the pooling kernel. If this is not specified, the default specified in the constructor is used.
- **strides** (*list, optional*) – Stride along width/height. If this is not specified, the default specified in the constructor is used.

Returns Pooled sequence.

Return type dynet.Expression

__init__(*default_kernel_size=None*, *default_strides=None*)
Initialize self. See help(type(self)) for accurate signature.

dynn.layers.pooling_layers.**max_pool_dim**(*x*, *d=0*, *kernel_width=None*, *stride=1*)
Efficent max pooling on GPU, assuming x is a matrix or a list of vectors

Recurrent layers

The particularity of recurrent is that their output can be fed back as input. This includes common recurrent cells like the Elman RNN or the LSTM.

class dynn.layers.recurrent_layers.**ElmanRNN**(*pc*, *input_dim*, *hidden_dim*, *activation=<function tanh>*, *dropout=0.0*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*, *dynn.layers.recurrent_layers.RecurrentCell*

The standard Elman RNN cell:

$h_t = \sigma(W_{hh}h_{t-1} + W_{hx}x_t + b)$

Parameters

- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters
- **input_dim** (*int*) – Input dimension
- **output_dim** (*int*) – Output (hidden) dimension
- **activation** (*function, optional*) – Activation function $sigma$ (default: *dynn.activations.tanh()*)
- **dropout** (*float, optional*) – Dropout rate (default 0)

__call__(*x*, *h*)
Perform the recurrent update.

Parameters

- **x** (dynet.Expression) – Input vector
- **h** (dynet.Expression) – Previous recurrent vector

Returns

Next recurrent state $h_t = \sigma(W_{hh}h_{t-1} + W_{hx}x_t + b)$

Return type dynet.Expression

__init__(*pc*, *input_dim*, *hidden_dim*, *activation=<function tanh>*, *dropout=0.0*)
Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)
Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to True, dropout is not applied (default: True)
- **update** (*bool, optional*) – Whether to update the parameters (default: True)

initial_value(*batch_size=1*)
Return a vector of dimension *hidden_dim* filled with zeros

Returns Zero vector

Return type dynet.Expression

class dynn.layers.recurrent_layers.**LSTM**(*pc*, *input_dim*, *hidden_dim*, *dropout_x=0.0*, *dropout_h=0.0*)
Bases: *dynn.layers.base_layers.ParametrizedLayer*, *dynn.layers.recurrent_layers.RecurrentCell*

Standard LSTM

Parameters

- **pc** (dynet.ParameterCollection) – Parameter collection to hold the parameters
- **input_dim** (*int*) – Input dimension
- **output_dim** (*int*) – Output (hidden) dimension
- **dropout_x** (*float, optional*) – Input dropout rate (default 0)
- **dropout_h** (*float, optional*) – Recurrent dropout rate (default 0)

__call__(*x*, *h*, *c*)
Perform the recurrent update.

Parameters

- **x** (dynet.Expression) – Input vector
- **h** (dynet.Expression) – Previous recurrent vector
- **c** (dynet.Expression) – Previous cell state vector

Returns

dynet.Expression for the ext recurrent states h and c

Return type tuple

__init__(*pc*, *input_dim*, *hidden_dim*, *dropout_x=0.0*, *dropout_h=0.0*)
Creates a subcollection for this layer with a custom name

init(*test=False*, *update=True*)
Initialize the layer before performing computation

Parameters

- **test** (*bool, optional*) – If test mode is set to True, dropout is not applied (default: True)

- **update** (*bool, optional*) – Whether to update the parameters (default: `True`)

initial_value(*batch_size=1*)
Return two vectors of dimension *hidden_dim* filled with zeros

Returns two zero vectors for h_0 and c_0

Return type tuple

class dynn.layers.recurrent_layers.**RecurrentCell**(**args*, ***kwargs*)
Bases: `object`

Base recurrent cell interface

Recurrent cells must provide a default initial value for their recurrent state (eg. all zeros)

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

initial_value(*batch_size=1*)
Initial value of the recurrent state. Should return a list.

Residual layers

class dynn.layers.residual_layers.**ResidualLayer**(*layer*, *shortcut_transform=None*, *layer_weight=1.0*, *shortcut_weight=1.0*)
Bases: *dynn.layers.base_layers.BaseLayer*

Adds residual connections to a layer

__call__(**args*, ***kwargs*)
Execute forward pass

__init__(*layer*, *shortcut_transform=None*, *layer_weight=1.0*, *shortcut_weight=1.0*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Initialize the layer before performing computation

For example setup dropout, freeze some parameters, etc...

Sequence transduction layers

Sequence transduction layers take in a sequence of expression and runs one layer over each input. They can be feed-forward (each input is treated independently, eg. *FeedForwardTransductionLayer*) or recurrent (the output at one step depends on the output at the previous step, eg. *UnidirectionalLayer*).

class dynn.layers.transduction_layers.**BidirectionalLayer**(*forward_cell*, *backward_cell*)
Bases: *dynn.layers.base_layers.BaseLayer*

Bidirectional transduction layer

This layer runs a recurrent cell on in each direction on a sequence of inputs and produces resulting the sequence of recurrent states.

Example:

```
# Parameter collection
pc = dy.ParameterCollection()
# LSTM cell
fwd_lstm_cell = dynn.layers.LSTM(pc, 10, 10)
bwd_lstm_cell = dynn.layers.LSTM(pc, 10, 10)
# Transduction layer
bilstm = dynn.layers.BidirectionalLayer(fwd_lstm_cell, bwd_lstm_cell)
# Inputs
dy.renew_cg()
xs = [dy.random_uniform(10, batch_size=5) for _ in range(20)]
# Initialize layer
bilstm.init(test=False)
# Transduce forward
fwd_states, bwd_states = bilstm(xs)
# Retrieve last h
fwd_h_final = fwd_states[-1][0]
# For the backward LSTM the final state is at
# the beginning of the sequence (assuming left padding)
bwd_h_final = fwd_states[0][0]
```

Parameters

- **forward_cell** (recurrent_layers.RecurrentCell) – The recurrent cell to use for forward transduction
- **backward_cell** (recurrent_layers.RecurrentCell) – The recurrent cell to use for backward transduction

__call__ (*input_sequence*, *lengths=None*, *left_padded=True*)

Transduces the sequence in both directions

The output is a tuple `forward_states, backward_states` where each `forward_states` is a list of the output states of the forward recurrent cell at each step (and `backward_states` for the backward cell). For instance in a BiLSTM the output is `[(fwd_h1, fwd_c1), ...], [(bwd_h1, bwd_c1), ...]`

This assumes that all the input expression have the same batch size. If you batch sentences of the same length together you should pad to the longest sequence.

Parameters

- **input_sequence** (*list*) – Input as a list of dynet.Expression objects
- **lengths** (*list, optional*) – If the expressions in the sequence are batched, but have different lengths, this should contain a list of the sequence lengths (default: None)
- **left_padded** (*bool, optional*) – If the input sequences have different lengths they must be padded to the length of longest sequence. Use this to specify whether the sequence is left or right padded.

Returns

List of forward and backward recurrent states (depends on the recurrent layer)

Return type tuple

__init__ (*forward_cell*, *backward_cell*)

Initialize self. See help(type(self)) for accurate signature.

init (**args*, ***kwargs*)

Passes its arguments to the recurrent layers

class dynn.layers.transduction_layers.**FeedForwardTransductionLayer**(*layer*)
Bases: *dynn.layers.base_layers.BaseLayer*

Feed forward transduction layer

This layer runs one cell on a sequence of inputs and returns the list of outputs. Calling it is equivalent to calling:

```
[layer(x) for x in input_sequence]
```

Parameters **cell** (base_layers.BaseLayer) – The recurrent cell to use for transduction

__call__(*input_sequence*)
Runs the layer over the input

The output is a list of the output of the layer at each step

Parameters **input_sequence** (*list*) – Input as a list of dynet.Expression objects

Returns List of recurrent states (depends on the recurrent layer)

Return type list

__init__(*layer*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Passes its arguments to the recurrent layer

class dynn.layers.transduction_layers.**UnidirectionalLayer**(*cell*)
Bases: *dynn.layers.base_layers.BaseLayer*

Unidirectional transduction layer

This layer runs a recurrent cell on a sequence of inputs and produces resulting the sequence of recurrent states.

Example:

```
# LSTM cell
lstm_cell = dynn.layers.LSTM(dy.ParameterCollection(), 10, 10)
# Transduction layer
lstm = dynn.layers.UnidirectionalLayer(lstm_cell)
# Inputs
dy.renew_cg()
xs = [dy.random_uniform(10, batch_size=5) for _ in range(20)]
# Initialize layer
lstm.init(test=False)
# Transduce forward
states = lstm(xs)
# Retrieve last h
h_final = states[-1][0]
```

Parameters **cell** (recurrent_layers.RecurrentCell) – The recurrent cell to use for transduction

__call__(*input_sequence*, *backward=False*, *lengths=None*, *left_padded=True*)
Transduces the sequence using the recurrent cell.

The output is a list of the output states at each step. For instance in an LSTM the output is `(h1, c1), (h2, c2), ...`

This assumes that all the input expression have the same batch size. If you batch sentences of the same length together you should pad to the longest sequence.

Parameters

- **input_sequence** (*list*) – Input as a list of dynet.Expression objects
- **backward** (*bool, optional*) – If this is True the sequence will be processed from left to right. The output sequence will still be in the same order as the input sequence though.
- **lengths** (*list, optional*) – If the expressions in the sequence are batched, but have different lengths, this should contain a list of the sequence lengths (default: None)
- **left_padded** (*bool, optional*) – If the input sequences have different lengths they must be padded to the length of longest sequence. Use this to specify whether the sequence is left or right padded.

Returns List of recurrent states (depends on the recurrent layer)

Return type list

__init__(*cell*)
Initialize self. See help(type(self)) for accurate signature.

init(**args*, ***kwargs*)
Passes its arguments to the recurrent layer

1.1.3 Submodules

1.1.3.1 Activation functions

Common activation functions for neural networks.

Most of those are wrappers around standard dynet operations (eg. rectify -> relu)

dynn.activations.**identity**(*x*)
The identity function

$y = x$

Parameters **x** (dynet.Expression) – Input expression

Returns x

Return type dynet.Expression

dynn.activations.**relu**(*x*)
The REctified Linear Unit

$y = \max(0, x)$

Parameters **x** (dynet.Expression) – Input expression

Returns $\max(0, x)$

Return type dynet.Expression

dynn.activations.**sigmoid**(*x*)
The sigmoid function

$y = \frac{1}{1+e^{-x}}$

Parameters **x** (dynet.Expression) – Input expression

Returns $\frac{1}{1+e^{-x}}$

Return type dynet.Expression

dynn.activations.**tanh**(*x*)
The hyperbolic tangent function

$y = \tanh(x)$

Parameters **x** (dynet.Expression) – Input expression

Returns $\tanh(x)$

Return type dynet.Expression

1.1.3.2 Operations

This extends the base dynet library with useful operations.

dynn.operations.**squeeze**(*x*, *d=0*)
Removes a dimension of size 1 at the given position

Example:

```
# (1, 20)
x = dy.zeros((1, 20))
# (20,)
squeeze(x, 0)
# (20, 1)
x = dy.zeros((20, 1))
# (20,)
squeeze(x, 1)
# (20,)
squeeze(x, -1)
```

dynn.operations.**unsqueeze**(*x*, *d=0*)
Insert a dimension of size 1 at the given position

Example:

```
# (10, 20)
x = dy.zeros((10, 20))
# (1, 10, 20)
unsqueeze(x, 0)
# (10, 20, 1)
unsqueeze(x, -1)
```

1.1.3.3 Parameter initialization

Some of those are just less verbose versions of dynet's PyInitializer s

dynn.parameter_initialization.**NormalInit**(*mean=0*, *std=1*)
Gaussian initialization

Parameters

- **mean** (*int, optional*) – Mean (default: 0.0)
- **std** (*int, optional*) – Standard deviation ($\neq$ variance) (default: 1.0)

Returns dy.NormalInitializer(mean, sqrt(std))

Return type dynet.PyInitializer

dynn.parameter_initialization.**OneInit**()
Initialize with 1

Returns dy.ConstInitializer(1)

Return type dynet.PyInitializer

dynn.parameter_initialization.**UniformInit**(*scale=1.0*)
Uniform initialization between -scale and scale

Parameters **scale** (*float*) – Scale of the distribution

Returns dy.UniformInitializer(scale)

Return type dynet.PyInitializer

dynn.parameter_initialization.**ZeroInit**()
Initialize with 0

Returns dy.ConstInitializer(0)

Return type dynet.PyInitializer

1.1.3.4 Utility functions

dynn.util.**conditional_dropout**(*x*, *dropout_rate*, *flag*)
This helper function applies dropout only if the flag is set to True and the dropout_rate is positive.

Parameters

- **x** (dynet.Expression) – Input expression
- **dropout_rate** (*float*) – Dropout rate
- **flag** (*bool*) – Setting this to false ensures that dropout is never applied (for testing for example)

dynn.util.**image_to_matrix**(*M*)
Transforms an 'image' with one channel (d1, d2, 1) into a matrix (d1, d2)

dynn.util.**list_to_matrix**(*l*)
Transforms a list of N vectors of dimension d into a (d, N) matrix

dynn.util.**mask_batches**(*x*, *mask*, *mode='mul'*)
Apply a mask to the batch dimension

Parameters

- **x** (list, dynet.Expression) – The expression we want to mask. Either a dynet.Expression or a list thereof with the same batch dimension.
- **mask** (np.array, list, dynet.Expression) – The mask. Either a list, 1d numpy array or dynet.Expression.
- **mode** (*str*) – One of "mul" and "add" for multiplicative and additive masks respectively

dynn.util.**matrix_to_image**(*M*)
Transforms a matrix (d1, d2) into an 'image' with one channel (d1, d2, 1)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

Z