conx Documentation

Release 3.0.3

Douglas Blank

Aug 07, 2017

Contents:

CHAPTER 1

conx

conx package

Submodules

conx.layers module

The conx.layers module contains the code for all of the layers. In addition, it dynamically loads all of the Keras layers and wraps them as a conx layer.

class conx.layers.**ActivationLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

ActivationLayer

Applies an activation function to an output.

Arguments

- **activation**: name of activation function to use (see: activations), or alternatively, a Theano or TensorFlow operation.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

CLASS
alias of Activation

class conx.layers.**ActivityRegularizationLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ActivityRegularizationLayer

Layer that applies an update to the cost function based input activity.

Arguments

- **l1**: L1 regularization factor (positive float).
- **l2**: L2 regularization factor (positive float).

Input shape

Arbitrary. Use the keyword argument input_shape
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

CLASS
alias of ActivityRegularization

class conx.layers.**AddLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AddLayer

Layer that adds a list of inputs.

It takes as input a list of tensors,
all of the same shape, and returns
a single tensor (also of the same shape).

CLASS
alias of Add

class conx.layers.**AlphaDropoutLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AlphaDropoutLayer

Applies Alpha Dropout to the input.

Alpha Dropout is a Dropout that keeps mean and variance of inputs
to their original values, in order to ensure the self-normalizing property
even after this dropout.
Alpha Dropout fits well to Scaled Exponential Linear Units
by randomly setting activations to the negative saturation value.

Arguments

- **rate**: float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.
- **seed**: A Python integer to use as random seed.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

References

- Self-Normalizing Neural Networks

CLASS
alias of `AlphaDropout`

class `conx.layers.`**`AverageLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AverageLayer

Layer that averages a list of inputs.

It takes as input a list of tensors,
all of the same shape, and returns
a single tensor (also of the same shape).

CLASS
alias of `Average`

class `conx.layers.`**`AveragePooling1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AveragePooling1DLayer

Average pooling for temporal data.

Arguments

- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

CLASS
alias of `AveragePooling1D`

class `conx.layers.`**`AveragePooling2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AveragePooling2DLayer

Average pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

CLASS
alias of `AveragePooling2D`

class `conx.layers.`**`AveragePooling3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

AveragePooling3DLayer

Average pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

CLASS

alias of `AveragePooling3D`

class `conx.layers.`**`AvgPool1DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

AvgPool1DLayer

Average pooling for temporal data.

Arguments

- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

CLASS

alias of `AveragePooling1D`

class `conx.layers.`**`AvgPool2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

AvgPool2DLayer

Average pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

CLASS

alias of `AveragePooling2D`

class conx.layers.**AvgPool3DLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

AvgPool3DLayer

Average pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`

- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

CLASS
alias of `AveragePooling3D`

class `conx.layers.`**BaseLayer**(*name*, **args*, ***params*)
Bases: `object`

The base class for all conx layers.

ACTIVATION_FUNCTIONS = ('relu', 'sigmoid', 'linear', 'softmax', 'tanh')

CLASS = None

get_minmax(*vector*)
Get the min/max for an input vector to this layer. Attempts to guess based on activation function.

kind()
Determines whether a layer is a "input", "hidden", or "output" layer based on its connections. If no connections, then it is "unconnected".

make_dummy_vector()
This is in the easy to use human format (list of lists ...)

make_image(*vector*, *config={}*)
Given an activation name (or function), and an output vector, display make and return an image widget.

make_input_layer_k()
Make an input layer for this type of layer. This allows Layers to have special kinds of input layers. Would need to be overrided in subclass.

make_keras_function()
This makes the Keras function for the functional interface.

make_keras_functions()
Make all Keras functions for this layer, including its own, dropout, etc.

scale_output_for_image(*vector*, *minmax*)
Given an activation name (or something else) and an output vector, scale the vector.

summary()
Print out a representation of the layer.

tooltip()
String (with newlines) for describing layer."

class `conx.layers.`**BatchNormalizationLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

BatchNormalizationLayer

Batch normalization layer (Ioffe and Szegedy, 2014).

Normalize the activations of the previous layer at each batch,
i.e. applies a transformation that maintains the mean activation
close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving average.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.
- **beta_regularizer**: Optional regularizer for the beta weight.
- **gamma_regularizer**: Optional regularizer for the gamma weight.
- **beta_constraint**: Optional constraint for the beta weight.
- **gamma_constraint**: Optional constraint for the gamma weight.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

References

- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

CLASS
alias of `BatchNormalization`

class `conx.layers.`**`BidirectionalLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

BidirectionalLayer

Bidirectional wrapper for RNNs.

Arguments

- **layer**: `Recurrent` instance.
- **merge_mode**: Mode by which outputs of the forward and backward RNNs will be combined. One of {'sum', 'mul', 'concat', 'ave', None}. If None, the outputs will not be combined, they will be returned as a list.

Raises

- **ValueError**: In case of invalid `merge_mode` argument.

Examples

```
model = Sequential()
model.add(Bidirectional(LSTM(10, return_sequences=True),
            input_shape=(5, 10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

CLASS
alias of Bidirectional

class conx.layers.**ConcatenateLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ConcatenateLayer

Layer that concatenates a list of inputs.

It takes as input a list of tensors,
all of the same shape expect for the concatenation axis,
and returns a single tensor, the concatenation of all inputs.

Arguments

- **axis**: Axis along which to concatenate.
- __**kwargs__: standard layer keyword arguments.

CLASS
alias of Concatenate

class conx.layers.**Conv1DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Conv1DLayer

1D convolution layer (e.g. temporal convolution).

This layer creates a convolution kernel that is convolved
with the layer input over a single spatial (or temporal) dimension
to produce a tensor of outputs.
If use_bias is True, a bias vector is created and added to the outputs.
Finally, if activation is not None,
it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide an input_shape argument
(tuple of integers or None, e.g.
(10, 128) for sequences of 10 vectors of 128-dimensional vectors,
or (None, 128) for variable-length sequences of 128-dimensional vectors.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"`, `"causal"` or `"same"` (case-insensitive). `"valid"` means "no padding". `"same"` results in padding the input such that the output has the same length as the original input. `"causal"` results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t+1:]. Useful when modeling temporal data where the model should not violate the temporal order. See WaveNet: A Generative Model for Raw Audio, section 2.1.
- **dilation_rate**: an integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

3D tensor with shape: `(batch_size, steps, input_dim)`

Output shape

3D tensor with shape: `(batch_size, new_steps, filters)`
`steps` value might have changed due to padding or strides.

CLASS

alias of `Conv1D`

class `conx.layers.`**`Conv2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

Conv2DLayer

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved
with the layer input to produce a tensor of

outputs. If `use_bias` is True,
a bias vector is created and added to the outputs. Finally, if
`activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(samples, channels, rows, cols)` if data_format='channels_first'

or 4D tensor with shape:
`(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(samples, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

CLASS
alias of `Conv2D`

class `conx.layers.`**`Conv2DTransposeLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Conv2DTransposeLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with

shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv2DTranspose`

class `conx.layers.`**`Conv3DLayer`**(*name*, **args*, ***params*)
Bases: *`conx.layers.BaseLayer`*

Conv3DLayer

3D convolution layer (e.g. spatial convolution over volumes).

This layer creates a convolution kernel that is convolved
with the layer input to produce a tensor of
outputs. If `use_bias` is True,
a bias vector is created and added to the outputs. Finally, if
`activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 128, 1)` for 128x128x128 volumes
with a single channel,
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 3 integers, specifying the depth, height and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along each spatial dimension. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).

•**bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape:
`(samples, channels, conv_dim1, conv_dim2, conv_dim3)` if data_format='channels_first'
or 5D tensor with shape:
`(samples, conv_dim1, conv_dim2, conv_dim3, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape:
`(samples, filters, new_conv_dim1, new_conv_dim2, new_conv_dim3)` if data_format='channels_first'
or 5D tensor with shape:
`(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, filters)` if data_format='channels_last'.
`new_conv_dim1`, `new_conv_dim2` and `new_conv_dim3` values might have changed due to padding.

CLASS
alias of `Conv3D`

class `conx.layers.`**Conv3DTransposeLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Conv3DTransposeLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 128, 3)` for a 128x128x128 volume with 3 channels
if `data_format="channels_last"`.

Arguments

•**filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

- **kernel_size**: An integer or tuple/list of 3 integers, specifying the width and height of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, depth, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, depth, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape:
`(batch, channels, depth, rows, cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, depth, rows, cols, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape:
`(batch, filters, new_depth, new_rows, new_cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, new_depth, new_rows, new_cols, filters)` if data_format='channels_last'.
`depth` and `rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv3DTranspose`

class `conx.layers.`**`ConvLSTM2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ConvLSTM2DLayer

Convolutional LSTM.

It is similar to an LSTM layer, but the input transformations
and recurrent transformations are both convolutional.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of n integers, specifying the dimensions of the convolution window.
- **strides**: An integer or tuple/list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, time, ..., channels)` while `channels_first` corresponds to inputs with shape `(batch, time, channels, ...)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: An integer or tuple/list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).

- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **go_backwards**: Boolean (default False). If True, rocess the input sequence backwards.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

Input shape

- if data_format='channels_first' 5D tensor with shape: `(samples,time, channels, rows, cols)`
- if data_format='channels_last' 5D tensor with shape: `(samples,time, rows, cols, channels)`

Output shape

- if `return_sequences`
  - if data_format='channels_first' 5D tensor with shape: `(samples, time, filters, output_row, output_col)`
  - if data_format='channels_last' 5D tensor with shape: `(samples, time, output_row, output_col, filters)`
- else
  - if data_format ='channels_first' 4D tensor with shape: `(samples, filters, output_row, output_col)`
  - if data_format='channels_last' 4D tensor with shape: `(samples, output_row, output_col, filters)` where o_row and o_col depend on the shape of the filter and the padding

Raises

- **ValueError**: in case of invalid constructor arguments.

References

- Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting The current implementation does not include the feedback loop on the cells output

CLASS

alias of `ConvLSTM2D`

class `conx.layers.`**`ConvRecurrent2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

ConvRecurrent2DLayer

Abstract base class for convolutional recurrent layers.

Do not use in a model – it's not a functional layer!

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of n integers, specifying the dimensions of the convolution window.
- **strides**: An integer or tuple/list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, time, ..., channels)` while `channels_first` corresponds to inputs with shape `(batch, time, channels, ...)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: An integer or tuple/list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **go_backwards**: Boolean (default False). If True, rocess the input sequence backwards.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

Input shape

5D tensor with shape `(num_samples, timesteps, channels, rows, cols)`.

Output shape

- if `return_sequences`: 5D tensor with shape `(num_samples, timesteps, channels, rows, cols)`.
- else, 4D tensor with shape `(num_samples, channels, rows, cols)`.

Masking

This layer supports masking for input data with a variable number
of timesteps. To introduce masks to your data,
use an Embedding layer with the `mask_zero` parameter
set to `True`.

- __**Note__:** for the time being, masking is only supported with Theano.

Note on using statefulness in RNNs

You can set RNN layers to be 'stateful', which means that the states
computed for the samples in one batch will be reused as initial states
for the samples in the next batch.
This assumes a one-to-one mapping between
samples in different successive batches.

To enable statefulness:
- specify `stateful=True` in the layer constructor.
- specify a fixed batch size for your model, by passing
a `batch_input_size=(...)` to the first layer in your model.
This is the expected shape of your inputs *including the batch size*.
It should be a tuple of integers, e.g. `(32, 10, 100)`.

To reset the states of your model, call `.reset_states()` on either
a specific layer, or on your entire model.

CLASS
alias of `ConvRecurrent2D`

class `conx.layers.`**`Convolution1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Convolution1DLayer

1D convolution layer (e.g. temporal convolution).

This layer creates a convolution kernel that is convolved
with the layer input over a single spatial (or temporal) dimension
to produce a tensor of outputs.
If `use_bias` is True, a bias vector is created and added to the outputs.
Finally, if `activation` is not `None`,
it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide an `input_shape` argument
(tuple of integers or `None`, e.g.
`(10, 128)` for sequences of 10 vectors of 128-dimensional vectors,
or `(None, 128)` for variable-length sequences of 128-dimensional vectors.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: One of `"valid"`, `"causal"` or `"same"` (case-insensitive). `"valid"` means "no padding". `"same"` results in padding the input such that the output has the same length as the original input. `"causal"` results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t+1:]. Useful when modeling temporal data where the model should not violate the temporal order. See WaveNet: A Generative Model for Raw Audio, section 2.1.

- **dilation_rate**: an integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

3D tensor with shape: `(batch_size, steps, input_dim)`

Output shape

3D tensor with shape: `(batch_size, new_steps, filters)`
`steps` value might have changed due to padding or strides.

CLASS

alias of `Conv1D`

class `conx.layers.`**`Convolution2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

Convolution2DLayer

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved
with the layer input to produce a tensor of
outputs. If `use_bias` is True,
a bias vector is created and added to the outputs. Finally, if
`activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(samples, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(samples, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

CLASS
alias of Conv2D

class conx.layers.**Convolution2DTransposeLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Convolution2DTransposeLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).

- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv2DTranspose`

class `conx.layers.`**`Convolution3DLayer`**(*name*, **args*, ***params*)
Bases: *`conx.layers.BaseLayer`*

Convolution3DLayer

3D convolution layer (e.g. spatial convolution over volumes).

This layer creates a convolution kernel that is convolved
with the layer input to produce a tensor of
outputs. If `use_bias` is True,
a bias vector is created and added to the outputs. Finally, if
`activation` is not `None`, it is applied to the outputs as well.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 128, 1)` for 128x128x128 volumes
with a single channel,

in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 3 integers, specifying the depth, height and width of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along each spatial dimension. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape:
`(samples, channels, conv_dim1, conv_dim2, conv_dim3)` if data_format='channels_first'
or 5D tensor with shape:
`(samples, conv_dim1, conv_dim2, conv_dim3, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape:

(samples, filters, new_conv_dim1, new_conv_dim2, new_conv_dim3) if data_format='channels_first'

or 5D tensor with shape:

(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, filters) if data_format='channels_last'.

new_conv_dim1, new_conv_dim2 and new_conv_dim3 values might have changed due to padding.

CLASS

alias of Conv3D

class conx.layers.**Cropping1DLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

Cropping1DLayer

Cropping layer for 1D input (e.g. temporal sequence).

It crops along the time dimension (axis 1).

Arguments

- **cropping**: int or tuple of int (length 2) How many units should be trimmed off at the beginning and end of the cropping dimension (axis 1). If a single int is provided, the same value will be used for both.

Input shape

3D tensor with shape (batch, axis_to_crop, features)

Output shape

3D tensor with shape (batch, cropped_axis, features)

CLASS

alias of Cropping1D

class conx.layers.**Cropping2DLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

Cropping2DLayer

Cropping layer for 2D input (e.g. picture).

It crops along spatial dimensions, i.e. width and height.

Arguments

- **cropping**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
  - If int: the same symmetric cropping is applied to width and height.
  - If tuple of 2 ints: interpreted as two different symmetric cropping values for height and width: (symmetric_height_crop, symmetric_width_crop).
  - If tuple of 2 tuples of 2 ints: interpreted as ((top_crop, bottom_crop), (left_crop, right_crop))
- **data_format**: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, cropped_rows, cropped_cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, cropped_rows, cropped_cols)`

Examples

```
# Crop the input 2D images or feature maps
model = Sequential()
model.add(Cropping2D(cropping=((2, 2), (4, 4)),
         input_shape=(28, 28, 3)))
# now model.output_shape == (None, 24, 20, 3)
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Cropping2D(cropping=((2, 2), (2, 2))))
# now model.output_shape == (None, 20, 16. 64)
```

CLASS
alias of `Cropping2D`

class `conx.layers.`**`Cropping3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Cropping3DLayer

Cropping layer for 3D data (e.g. spatial or spatio-temporal).

Arguments

- **cropping**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
  - If int: the same symmetric cropping is applied to width and height.
  - If tuple of 2 ints: interpreted as two different symmetric cropping values for height and width: `(symmetric_dim1_crop, symmetric_dim2_crop, symmetric_dim3_crop)`.
  - If tuple of 2 tuples of 2 ints: interpreted as `((left_dim1_crop, right_dim1_crop), (left_dim2_crop, right_dim2_crop), (left_dim3_crop, right_dim3_crop))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop, depth)`

- If `data_format` is `"channels_first"`: `(batch, depth, first_axis_to_crop, second_axis_to_crop, third_axis_to_crop)`

Output shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, first_cropped_axis, second_cropped_axis, third_cropped_axis, depth)`
- If `data_format` is `"channels_first"`: `(batch, depth, first_cropped_axis, second_cropped_axis, third_cropped_axis)`

CLASS

alias of `Cropping3D`

class `conx.layers.`**`Deconv2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

Deconv2DLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv2DTranspose`

class `conx.layers.`**`Deconv3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Deconv3DLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises

from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 128, 3)` for a 128x128x128 volume with 3 channels
if `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 3 integers, specifying the width and height of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, depth, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, depth, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape:
`(batch, channels, depth, rows, cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, depth, rows, cols, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape:
`(batch, filters, new_depth, new_rows, new_cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, new_depth, new_rows, new_cols, filters)` if data_format='channels_last'.
`depth` and `rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv3DTranspose`

class `conx.layers.`**`Deconvolution2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Deconvolution2DLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures
in `data_format="channels_last"`.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **dilation_rate**: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning

•Deconvolutional Networks

CLASS
alias of `Conv2DTranspose`

class `conx.layers.`**`Deconvolution3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

Deconvolution3DLayer

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises
from the desire to use a transformation going in the opposite direction
of a normal convolution, i.e., from something that has the shape of the
output of some convolution to something that has the shape of its input
while maintaining a connectivity pattern that is compatible with
said convolution.

When using this layer as the first layer in a model,
provide the keyword argument `input_shape`
(tuple of integers, does not include the sample axis),
e.g. `input_shape=(128, 128, 128, 3)` for a 128x128x128 volume with 3 channels
if `data_format="channels_last"`.

Arguments

•**filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

•**kernel_size**: An integer or tuple/list of 3 integers, specifying the width and height of the 3D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

•**strides**: An integer or tuple/list of 3 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.

•**padding**: one of `"valid"` or `"same"` (case-insensitive).

•**data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, depth, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, depth, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

•**dilation_rate**: an integer or tuple/list of 3 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any stride value != 1.

•**activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).

•**use_bias**: Boolean, whether the layer uses a bias vector.

•**kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).

•**bias_initializer**: Initializer for the bias vector (see initializers).

- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

5D tensor with shape:
`(batch, channels, depth, rows, cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, depth, rows, cols, channels)` if data_format='channels_last'.

Output shape

5D tensor with shape:
`(batch, filters, new_depth, new_rows, new_cols)` if data_format='channels_first'
or 5D tensor with shape:
`(batch, new_depth, new_rows, new_cols, filters)` if data_format='channels_last'.
`depth` and `rows` and `cols` values might have changed due to padding.

References

- A guide to convolution arithmetic for deep learning
- Deconvolutional Networks

CLASS
alias of `Conv3DTranspose`

`conx.layers.`**DenseLayer**
alias of *Layer*

class `conx.layers.`**DotLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

DotLayer

Layer that computes a dot product between samples in two tensors.

E.g. if applied to two tensors `a` and `b` of shape `(batch_size, n)`,
the output will be a tensor of shape `(batch_size, 1)`
where each entry `i` will be the dot product between
`a[i]` and `b[i]`.

Arguments

- **axes**: Integer or tuple of integers, axis or axes along which to take the dot product.

- **normalize**: Whether to L2-normalize samples along the dot product axis before taking the dot product. If set to True, then the output of the dot product is the cosine proximity between the two samples.
- __**kwargs__: Standard layer keyword arguments.

CLASS
alias of `Dot`

class `conx.layers.`**`DropoutLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

DropoutLayer

Applies Dropout to the input.

Dropout consists in randomly setting
a fraction `rate` of input units to 0 at each update during training time,
which helps prevent overfitting.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.
- **noise_shape**: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- **seed**: A Python integer to use as random seed.

References

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

CLASS
alias of `Dropout`

class `conx.layers.`**`ELULayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ELULayer

Exponential Linear Unit.

It follows:
`f(x) = alpha * (exp(x) - 1.) for x < 0,`
`f(x) = x for x >= 0.`

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as the input.

Arguments

- **alpha**: scale for the negative factor.

References

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

CLASS
alias of ELU

class conx.layers.**EmbeddingLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

EmbeddingLayer

Turns positive integers (indexes) into dense vectors of fixed size.
eg. [[4], [20]] -> [[0.25, 0.1], [0.6, -0.2]]

This layer can only be used as the first layer in a model.

Example

```
model = Sequential()
model.add(Embedding(1000, 64, input_length=10))
# the model will take as input an integer matrix of size (batch, input_length).
# the largest integer (i.e. word index) in the input should be no larger than 999␣
↪(vocabulary size).
# now model.output_shape == (None, 10, 64), where None is the batch dimension.

input_array = np.random.randint(1000, size=(32, 10))

model.compile('rmsprop', 'mse')
output_array = model.predict(input_array)
assert output_array.shape == (32, 10, 64)
```

Arguments

- **input_dim**: int > 0. Size of the vocabulary, i.e. maximum integer index + 1.
- **output_dim**: int >= 0. Dimension of the dense embedding.
- **embeddings_initializer**: Initializer for the `embeddings` matrix (see initializers).
- **embeddings_regularizer**: Regularizer function applied to the `embeddings` matrix (see regularizer).
- **embeddings_constraint**: Constraint function applied to the `embeddings` matrix (see constraints).
- **mask_zero**: Whether or not the input value 0 is a special "padding" value that should be masked out. This is useful when using recurrent layers which may take variable length input. If this is `True` then all subsequent layers in the model need to support masking or an exception will be raised. If mask_zero is set to True, as a consequence, index 0 cannot be used in the vocabulary (input_dim should equal size of vocabulary + 1).
- **input_length**: Length of input sequences, when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed).

Input shape

2D tensor with shape: `(batch_size, sequence_length)`.

Output shape

3D tensor with shape: `(batch_size, sequence_length, output_dim)`.

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

CLASS
alias of `Embedding`

class `conx.layers.`**`FlattenLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

FlattenLayer

Flattens the input. Does not affect the batch size.

Example

```
model = Sequential()
model.add(Conv2D(64, 3, 3,
            border_mode='same',
            input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

CLASS
alias of `Flatten`

class `conx.layers.`**`GRULayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GRULayer

Gated Recurrent Unit - Cho et al. 2014.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations). If you pass None, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).

- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

References

- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

CLASS
alias of `GRU`

class `conx.layers.`**`GaussianDropoutLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GaussianDropoutLayer

Apply multiplicative 1-centered Gaussian noise.

As it is a regularization layer, it is only active at training time.

Arguments

- **rate**: float, drop probability (as with `Dropout`). The multiplicative noise will have standard deviation `sqrt(rate / (1 - rate))`.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

References

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting Srivastava, Hinton, et al. 2014

CLASS
alias of `GaussianDropout`

class `conx.layers.`**`GaussianNoiseLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GaussianNoiseLayer

Apply additive zero-centered Gaussian noise.

This is useful to mitigate overfitting
(you could see it as a form of random data augmentation).
Gaussian Noise (GS) is a natural choice as corruption process
for real valued inputs.

As it is a regularization layer, it is only active at training time.

Arguments

- **stddev**: float, standard deviation of the noise distribution.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as input.

CLASS
alias of `GaussianNoise`

class `conx.layers.`**`GlobalAveragePooling1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAveragePooling1DLayer

Global average pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling1D`

class `conx.layers.`**`GlobalAveragePooling2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAveragePooling2DLayer

Global average pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling2D`

class `conx.layers.`**`GlobalAveragePooling3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAveragePooling3DLayer

Global Average pooling operation for 3D data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling3D`

class conx.layers.**GlobalAvgPool1DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAvgPool1DLayer

Global average pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling1D`

class conx.layers.**GlobalAvgPool2DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAvgPool2DLayer

Global average pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling2D`

class conx.layers.**GlobalAvgPool3DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalAvgPool3DLayer

Global Average pooling operation for 3D data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalAveragePooling3D`

class `conx.layers.`**`GlobalMaxPool1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPool1DLayer

Global max pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling1D`

class `conx.layers.`**`GlobalMaxPool2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPool2DLayer

Global max pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling2D`

class `conx.layers.`**`GlobalMaxPool3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPool3DLayer

Global Max pooling operation for 3D data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling3D`

class `conx.layers.`**`GlobalMaxPooling1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPooling1DLayer

Global max pooling operation for temporal data.

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling1D`

class `conx.layers.`**`GlobalMaxPooling2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPooling2DLayer

Global max pooling operation for spatial data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling2D`

class `conx.layers.`**`GlobalMaxPooling3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

GlobalMaxPooling3DLayer

Global Max pooling operation for 3D data.

Arguments

- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

2D tensor with shape:
`(batch_size, channels)`

CLASS
alias of `GlobalMaxPooling3D`

class `conx.layers.`**`HighwayLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

HighwayLayer

Densely connected highway network.
Highway layers are a natural extension of LSTMs to feedforward networks.
Arguments

- **init**: name of initialization function for the weights of the layer (see initializations), or alternatively, Theano function to use for weights initialization. This parameter is only relevant if you don't pass a `weights` argument.
- **activation**: name of activation function to use (see activations), or alternatively, elementwise Theano function. If you don't specify anything, no activation is applied (ie. "linear" activation: a(x) = x).
- **weights**: list of Numpy arrays to set as initial weights. The list should have 2 elements, of shape `(input_dim, output_dim)` and (output_dim,) for weights and biases respectively.
- **W_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the main weights matrix.
- **b_regularizer**: instance of WeightRegularizer, applied to the bias.
- **activity_regularizer**: instance of ActivityRegularizer, applied to the network output.
- **W_constraint**: instance of the constraints module (eg. maxnorm, nonneg), applied to the main weights matrix.
- **b_constraint**: instance of the constraints module, applied to the bias.
- **bias**: whether to include a bias (i.e. make the layer affine rather than linear).
- **input_dim**: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model. **Input shape**

2D tensor with shape: `(nb_samples, input_dim)`.
Output shape

2D tensor with shape: `(nb_samples, input_dim)`.

References

- Highway Networks

CLASS
alias of `Highway`

`conx.layers.`**`InputLayer`**
alias of *Layer*

class `conx.layers.`**`InputLayerLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

InputLayerLayer

Layer to be used as an entry point into a graph.

It can either wrap an existing tensor (pass an `input_tensor` argument)
or create its a placeholder tensor (pass arguments `input_shape`
or `batch_input_shape` as well as `dtype`).

Arguments

- **input_shape**: Shape tuple, not including the batch axis.
- **batch_size**: Optional input batch size (integer or None).
- **batch_input_shape**: Shape tuple, including the batch axis.
- **dtype**: Datatype of the input.
- **input_tensor**: Optional tensor to use as layer input instead of creating a placeholder.
- **sparse**: Boolean, whether the placeholder created is meant to be sparse.
- **name**: Name of the layer (string).

CLASS
alias of *InputLayer*

class `conx.layers.`**`LSTMLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

LSTMLayer

Long-Short Term Memory unit - Hochreiter 1997.

For a step-by-step description of the algorithm, see
this tutorial.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations). If you pass None, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation**: Activation function to use for the recurrent step (see activations).

- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **unit_forget_bias**: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al.
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

References

- Long short-term memory (original 1997 paper)
- Learning to forget: Continual prediction with LSTM
- Supervised sequence labeling with recurrent neural networks
- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

CLASS
alias of `LSTM`

class `conx.layers.`**`LambdaLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

LambdaLayer

Wraps arbitrary expression as a `Layer` object.

Examples

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
```

```
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
         output_shape=antirectifier_output_shape))
```

Arguments

- **function**: The function to be evaluated. Takes input tensor as first argument.
- **output_shape**: Expected output shape from function. Only relevant when using Theano. Can be a tuple or function. If a tuple, it only specifies the first dimension onward; sample dimension is assumed either the same as the input: `output_shape = (input_shape[0], ) + output_shape` or, the input is `None` and the sample dimension is also `None`: `output_shape = (None, ) + output_shape` If a function, it specifies the entire shape as a function of the input shape: `output_shape = f(input_shape)`
- **arguments**: optional dictionary of keyword arguments to be passed to the function.

Input shape

Arbitrary. Use the keyword argument input_shape
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Specified by `output_shape` argument
(or auto-inferred when using TensorFlow).

CLASS
alias of `Lambda`

class `conx.layers.`**Layer**(*name*, *shape*, ***params*)
Bases: *conx.layers.BaseLayer*

For Dense and Input type layers.

CLASS
alias of `Dense`

make_keras_function()
For all Keras-based functions. Returns the Keras class.

summary()
Print a summary of the dense/input layer.

class conx.layers.**LayerLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

LayerLayer

Abstract base layer class.

Properties

- **name**: String, must be unique within a model.
- **input_spec**: List of InputSpec class instances each entry describes one required input:
  - ndim
  - dtype A layer with `n` input tensors must have an `input_spec` of length `n`.
- **trainable**: Boolean, whether the layer weights will be updated during training.
- **uses_learning_phase**: Whether any operation of the layer uses `K.in_training_phase()` or `K.in_test_phase()`.
- **input_shape**: Shape tuple. Provided for convenience, but note that there may be cases in which this attribute is ill-defined (e.g. a shared layer with multiple input shapes), in which case requesting `input_shape` will raise an Exception. Prefer using `layer.get_input_shape_for(input_shape)`, or `layer.get_input_shape_at(node_index)`.
- **output_shape**: Shape tuple. See above.
- **inbound_nodes**: List of nodes.
- **outbound_nodes**: List of nodes. input, output: Input/output tensor(s). Note that if the layer is used more than once (shared layer), this is ill-defined and will raise an exception. In such cases, use `layer.get_input_at(node_index)`. input_mask, output_mask: Same as above, for masks.
- **trainable_weights**: List of variables.
- **non_trainable_weights**: List of variables.
- **weights**: The concatenation of the lists trainable_weights and non_trainable_weights (in this order).
- **constraints**: Dict mapping weights to constraints.

Methods

call(x, mask=None): Where the layer's logic lives.
call(x, mask=None): Wrapper around the layer logic (`call`).
If x is a Keras tensor:
- Connect current layer with last layer from tensor:
`self._add_inbound_node(last_layer)`
- Add layer to tensor history
If layer is not built:
- Build from x._keras_shape
get_weights()
set_weights(weights)
get_config()
count_params()
compute_output_shape(input_shape)
compute_mask(x, mask)

get_input_at(node_index)
get_output_at(node_index)
get_input_shape_at(node_index)
get_output_shape_at(node_index)
get_input_mask_at(node_index)
get_output_mask_at(node_index)

Class Methods

from_config(config)

Internal methods:

build(input_shape)
_add_inbound_node(layer, index=0)
assert_input_compatibility()

CLASS
alias of *Layer*

class conx.layers.**LeakyReLULayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

LeakyReLULayer

Leaky version of a Rectified Linear Unit.

It allows a small gradient when the unit is not active:
`f(x) = alpha * x for x < 0,`
`f(x) = x for x >= 0.`

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as the input.

Arguments

•**alpha**: float >= 0. Negative slope coefficient.

References

•Rectifier Nonlinearities Improve Neural Network Acoustic Models

CLASS
alias of `LeakyReLU`

class conx.layers.**LocallyConnected1DLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

LocallyConnected1DLayer

Locally-connected layer for 1D inputs.

The LocallyConnected1D layer works similarly to
the Conv1D layer, except that weights are unshared,
that is, a different set of filters is applied at each different patch
of the input.

Example

```
# apply a unshared weight convolution 1d of length 3 to a sequence with
# 10 timesteps, with 64 output filters
model = Sequential()
model.add(LocallyConnected1D(64, 3, input_shape=(10, 32)))
# now model.output_shape == (None, 8, 64)
# add a new conv1d on top
model.add(LocallyConnected1D(32, 3))
# now model.output_shape == (None, 6, 32)
```

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or tuple/list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- **padding**: Currently only supports "valid" (case-insensitive). "same" may be supported in the future.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: a(x) = x).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the kernel weights matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the kernel weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

3D tensor with shape: (batch_size, steps, input_dim)

Output shape

3D tensor with shape: `(batch_size, new_steps, filters)`
`steps` value might have changed due to padding or strides.

CLASS
alias of `LocallyConnected1D`

class `conx.layers.`**LocallyConnected2DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

LocallyConnected2DLayer

Locally-connected layer for 2D inputs.

The `LocallyConnected2D` layer works similarly
to the `Conv2D` layer, except that weights are unshared,
that is, a different set of filters is applied at each
different patch of the input.

Examples

```
# apply a 3x3 unshared weights convolution with 64 output filters on a 32x32 image
# with `data_format="channels_last"`:
model = Sequential()
model.add(LocallyConnected2D(64, (3, 3), input_shape=(32, 32, 3)))
# now model.output_shape == (None, 30, 30, 64)
# notice that this layer will consume (30*30)*(3*3*3*64) + (30*30)*64 parameters

# add a 3x3 unshared weights convolution on top, with 32 output filters:
model.add(LocallyConnected2D(32, (3, 3)))
# now model.output_shape == (None, 28, 28, 32)
```

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions.
- **padding**: Currently only support `"valid"` (case-insensitive). `"same"` will be supported in future.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix (see initializers).

- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(samples, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(samples, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

CLASS
alias of `LocallyConnected2D`

class `conx.layers.`**MaskingLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaskingLayer

Masks a sequence by using a mask value to skip timesteps.

For each timestep in the input tensor (dimension #1 in the tensor),
if all values in the input tensor at that timestep
are equal to `mask_value`, then the timestep will be masked (skipped)
in all downstream layers (as long as they support masking).

If any downstream layer does not support masking yet receives such
an input mask, an exception will be raised.

Example

Consider a Numpy data array `x` of shape `(samples, timesteps, features)`,

to be fed to a LSTM layer.
You want to mask timestep #3 and #5 because you lack data for
these timesteps. You can:

- set `x[:, 3, :] = 0.` and `x[:, 5, :] = 0.`
- insert a `Masking` layer with `mask_value=0.` before the LSTM layer:

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```

CLASS
alias of `Masking`

class `conx.layers.`**`MaxPool1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxPool1DLayer

Max pooling operation for temporal data.

Arguments

- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

CLASS
alias of `MaxPooling1D`

class `conx.layers.`**`MaxPool2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxPool2DLayer

Max pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value

found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

CLASS

alias of `MaxPooling2D`

class `conx.layers.`**`MaxPool3DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

MaxPool3DLayer

Max pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

CLASS

alias of `MaxPooling3D`

class conx.layers.**MaxPooling1DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxPooling1DLayer

Max pooling operation for temporal data.

Arguments

- **pool_size**: Integer, size of the max pooling windows.
- **strides**: Integer, or None. Factor by which to downscale. E.g. 2 will halve the input. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).

Input shape

3D tensor with shape: `(batch_size, steps, features)`.

Output shape

3D tensor with shape: `(batch_size, downsampled_steps, features)`.

CLASS
alias of `MaxPooling1D`

class conx.layers.**MaxPooling2DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxPooling2DLayer

Max pooling operation for spatial data.

Arguments

- **pool_size**: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.
- **strides**: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, rows, cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, rows, cols)`

Output shape

- If `data_format='channels_last'`: 4D tensor with shape: `(batch_size, pooled_rows, pooled_cols, channels)`
- If `data_format='channels_first'`: 4D tensor with shape: `(batch_size, channels, pooled_rows, pooled_cols)`

CLASS
alias of MaxPooling2D

class conx.layers.**MaxPooling3DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxPooling3DLayer

Max pooling operation for 3D data (spatial or spatio-temporal).

Arguments

- **pool_size**: tuple of 3 integers, factors by which to downscale (dim1, dim2, dim3). (2, 2, 2) will halve the size of the 3D input in each dimension.
- **strides**: tuple of 3 integers, or None. Strides values.
- **padding**: One of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, spatial_dim1, spatial_dim2, spatial_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, spatial_dim1, spatial_dim2, spatial_dim3)`

Output shape

- If `data_format='channels_last'`: 5D tensor with shape: `(batch_size, pooled_dim1, pooled_dim2, pooled_dim3, channels)`
- If `data_format='channels_first'`: 5D tensor with shape: `(batch_size, channels, pooled_dim1, pooled_dim2, pooled_dim3)`

CLASS
alias of MaxPooling3D

class conx.layers.**MaximumLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaximumLayer

Layer that computes the maximum (element-wise) a list of inputs.

It takes as input a list of tensors,
all of the same shape, and returns
a single tensor (also of the same shape).

CLASS
alias of Maximum

class conx.layers.**MaxoutDenseLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MaxoutDenseLayer

A dense maxout layer.
A `MaxoutDense` layer takes the element-wise maximum of
`nb_feature` `Dense(input_dim, output_dim)` linear layers.
This allows the layer to learn a convex,
piecewise linear activation function over the inputs.
Note that this is a *linear* layer;
if you wish to apply activation function
(you shouldn't need to –they are universal function approximators),
an `Activation` layer must be added after.
Arguments

- **output_dim**: int > 0.
- **nb_feature**: number of Dense layers to use internally.
- **init**: name of initialization function for the weights of the layer (see initializations), or alternatively, Theano function to use for weights initialization. This parameter is only relevant if you don't pass a `weights` argument.
- **weights**: list of Numpy arrays to set as initial weights. The list should have 2 elements, of shape `(input_dim, output_dim)` and (output_dim,) for weights and biases respectively.
- **W_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the main weights matrix.
- **b_regularizer**: instance of WeightRegularizer, applied to the bias.
- **activity_regularizer**: instance of ActivityRegularizer, applied to the network output.
- **W_constraint**: instance of the constraints module (eg. maxnorm, nonneg), applied to the main weights matrix.
- **b_constraint**: instance of the constraints module, applied to the bias.
- **bias**: whether to include a bias (i.e. make the layer affine rather than linear).
- **input_dim**: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model. **Input shape**

2D tensor with shape: `(nb_samples, input_dim)`.
Output shape

2D tensor with shape: `(nb_samples, output_dim)`.
References

- Maxout Networks

CLASS
alias of `MaxoutDense`

class `conx.layers.`**`MergeLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MergeLayer

A `Merge` layer can be used to merge a list of tensors
into a single tensor, following some merge `mode`.
Example

```
model1 = Sequential()
model1.add(Dense(32, input_dim=32))
model2 = Sequential()
model2.add(Dense(32, input_dim=32))
merged_model = Sequential()
merged_model.add(Merge([model1, model2], mode='concat', concat_axis=1))
```

Arguments

- **layers**: Can be a list of Keras tensors or a list of layer instances. Must be more than one layer/tensor.
- **mode**: String or lambda/function. If string, must be one
  - **of**: 'sum', 'mul', 'concat', 'ave', 'cos', 'dot', 'max'. If lambda/function, it should take as input a list of tensors and return a single tensor.
- **concat_axis**: Integer, axis to use in mode `concat`.
- **dot_axes**: Integer or tuple of integers, axes to use in mode `dot` or `cos`.
- **output_shape**: Either a shape tuple (tuple of integers), or a lambda/function to compute `output_shape` (only if merge mode is a lambda/function). If the argument is a tuple, it should be expected output shape, *not* including the batch size (same convention as the `input_shape` argument in layers). If the argument is callable, it should take as input a list of shape tuples (1:1 mapping to input tensors) and return a single shape tuple, including the batch size (same convention as the `compute_output_shape` method of layers).
- **node_indices**: Optional list of integers containing the output node index for each input layer (in case some input layers have multiple output nodes). will default to an array of 0s if not provided.
- **tensor_indices**: Optional list of indices of output tensors to consider for merging (in case some input layer node returns multiple tensors).
- **output_mask**: Mask or lambda/function to compute the output mask (only if merge mode is a lambda/function). If the latter case, it should take as input a list of masks and return a single mask.

CLASS
alias of `Merge`

class `conx.layers.`**`MultiplyLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

MultiplyLayer

Layer that multiplies (element-wise) a list of inputs.

It takes as input a list of tensors,
all of the same shape, and returns
a single tensor (also of the same shape).

CLASS
alias of `Multiply`

class `conx.layers.`**`PReLULayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

PReLULayer

Parametric Rectified Linear Unit.

It follows:
`f(x) = alpha * x for x < 0`,
`f(x) = x for x >= 0`,
where `alpha` is a learned array with the same shape as x.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as the input.

Arguments

- **alpha_initializer**: initializer function for the weights.
- **alpha_regularizer**: regularizer for the weights.
- **alpha_constraint**: constraint for the weights.
- **shared_axes**: the axes along which to share learnable parameters for the activation function. For example, if the incoming feature maps are from a 2D convolution with output shape `(batch, height, width, channels)`, and you wish to share parameters across space so that each filter only has one set of parameters, set `shared_axes=[1, 2]`.

References

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

CLASS
alias of `PReLU`

class `conx.layers.`**`PermuteLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

PermuteLayer

Permutes the dimensions of the input according to a given pattern.

Useful for e.g. connecting RNNs and convnets together.

Example

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

Arguments

- **dims**: Tuple of integers. Permutation pattern, does not include the samples dimension. Indexing starts at 1. For instance, `(2, 1)` permutes the first and second dimension of the input.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same as the input shape, but with the dimensions re-ordered according
to the specified pattern.

CLASS

alias of `Permute`

class conx.layers.**PictureLayer**(*name*, *shape*, ***params*)

Bases: *conx.layers.Layer*

A class for pictures. WIP.

make_image(*vector*, *config={}*)

Given an activation name (or function), and an output vector, display make and return an image widget.

class conx.layers.**RecurrentLayer**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

RecurrentLayer

Abstract base class for recurrent layers.

Do not use in a model – it's not a valid layer!
Use its children classes `LSTM`, `GRU` and `SimpleRNN` instead.

All recurrent layers (`LSTM`, `GRU`, `SimpleRNN`) also
follow the specifications of this class and accept
the keyword arguments listed below.

Example

```
# as the first layer in a Sequential model
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
# now model.output_shape == (None, 32)
# note: `None` is the batch dimension.

# for subsequent layers, no need to specify the input size:
model.add(LSTM(16))

# to stack recurrent layers, you must use return_sequences=True
# on any recurrent layer that feeds into another recurrent layer.
# note that you only need to specify the input size on the first layer.
model = Sequential()
model.add(LSTM(64, input_dim=64, input_length=10, return_sequences=True))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(10))
```

Arguments

- **weights**: list of Numpy arrays to set as initial weights. The list should have 3 elements, of shapes: `[(input_dim, output_dim), (output_dim, output_dim), (output_dim,)]`.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **implementation**: one of {0, 1, or 2}. If set to 0, the RNN will use an implementation that uses fewer, larger matrix products, thus running faster on CPU but consuming more memory. If set to 1, the RNN will use more matrix products, but smaller ones, thus running slower (may actually be faster on GPU) while consuming less memory. If set to 2 (LSTM/GRU only), the RNN will combine the input gate, the forget gate and the output gate into a single matrix, enabling more time-efficient parallelization on the GPU.
  - **Note**: RNN dropout must be shared for all gates, resulting in a slightly reduced regularization.
- **input_dim**: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model.
- **input_length**: Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the `input_shape` argument)

Input shapes

3D tensor with shape `(batch_size, timesteps, input_dim)`,
(Optional) 2D tensors with shape `(batch_size, output_dim)`.

Output shape

- if `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, units)`.
- if `return_sequences`: 3D tensor with shape `(batch_size, timesteps, units)`.
- else, 2D tensor with shape `(batch_size, units)`.

Masking

This layer supports masking for input data with a variable number
of timesteps. To introduce masks to your data,
use an Embedding layer with the `mask_zero` parameter
set to `True`.

Note on using statefulness in RNNs

You can set RNN layers to be 'stateful', which means that the states
computed for the samples in one batch will be reused as initial states
for the samples in the next batch. This assumes a one-to-one mapping
between samples in different successive batches.

To enable statefulness:
- specify `stateful=True` in the layer constructor.
- specify a fixed batch size for your model, by passing
if sequential model:
`batch_input_shape=(...)` to the first layer in your model.
else for functional model with 1 or more Input layers:
`batch_shape=(...)` to all the first layers in your model.
This is the expected shape of your inputs
including the batch size.
It should be a tuple of integers, e.g. `(32, 10, 100)`.
- specify `shuffle=False` when calling fit().

To reset the states of your model, call `.reset_states()` on either
a specific layer, or on your entire model.

Note on specifying the initial state of RNNs

You can specify the initial state of RNN layers symbolically by
calling them with the keyword argument `initial_state`. The value of
`initial_state` should be a tensor or list of tensors representing
the initial state of the RNN layer.

You can specify the initial state of RNN layers numerically by

calling `reset_states` with the keyword argument `states`. The value of
`states` should be a numpy array or list of numpy arrays representing
the initial state of the RNN layer.

CLASS
alias of `Recurrent`

class `conx.layers.`**`RepeatVectorLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

RepeatVectorLayer

Repeats the input n times.

Example

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension

model.add(RepeatVector(3))
# now: model.output_shape == (None, 3, 32)
```

Arguments

- **n**: integer, repetition factor.

Input shape

2D tensor of shape `(num_samples, features)`.

Output shape

3D tensor of shape `(num_samples, n, features)`.

CLASS
alias of `RepeatVector`

class `conx.layers.`**`ReshapeLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ReshapeLayer

Reshapes an output to a certain shape.

Arguments

- **target_shape**: target shape. Tuple of integers. Does not include the batch axis.

Input shape

Arbitrary, although all dimensions in the input shaped must be fixed.
Use the keyword argument `input_shape`
(tuple of integers, does not include the batch axis)
when using this layer as the first layer in a model.

Output shape

`(batch_size,) + target_shape`

Example

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

CLASS

alias of `Reshape`

class `conx.layers.`**`SeparableConv2DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

SeparableConv2DLayer

Depthwise separable 2D convolution.

Separable convolutions consist in first performing
a depthwise spatial convolution
(which acts on each input channel separately)
followed by a pointwise convolution which mixes together the resulting
output channels. The `depth_multiplier` argument controls how many
output channels are generated per input channel in the depthwise step.

Intuitively, separable convolutions can be understood as
a way to factorize a convolution kernel into two smaller kernels,
or as an extreme version of an Inception block.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value

found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

- **depth_multiplier**: The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `filterss_in * depth_multiplier`.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **depthwise_initializer**: Initializer for the depthwise kernel matrix (see initializers).
- **pointwise_initializer**: Initializer for the pointwise kernel matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **depthwise_regularizer**: Regularizer function applied to the depthwise kernel matrix (see regularizer).
- **pointwise_regularizer**: Regularizer function applied to the depthwise kernel matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **depthwise_constraint**: Constraint function applied to the depthwise kernel matrix (see constraints).
- **pointwise_constraint**: Constraint function applied to the pointwise kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

CLASS
alias of `SeparableConv2D`

class `conx.layers.`**SeparableConvolution2DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

SeparableConvolution2DLayer

Depthwise separable 2D convolution.

Separable convolutions consist in first performing
a depthwise spatial convolution
(which acts on each input channel separately)
followed by a pointwise convolution which mixes together the resulting
output channels. The `depth_multiplier` argument controls how many
output channels are generated per input channel in the depthwise step.

Intuitively, separable convolutions can be understood as
a way to factorize a convolution kernel into two smaller kernels,
or as an extreme version of an Inception block.

Arguments

- **filters**: Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the width and height. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of `"valid"` or `"same"` (case-insensitive).
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".
- **depth_multiplier**: The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to `filterss_in * depth_multiplier`.
- **activation**: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **depthwise_initializer**: Initializer for the depthwise kernel matrix (see initializers).
- **pointwise_initializer**: Initializer for the pointwise kernel matrix (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **depthwise_regularizer**: Regularizer function applied to the depthwise kernel matrix (see regularizer).
- **pointwise_regularizer**: Regularizer function applied to the depthwise kernel matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **depthwise_constraint**: Constraint function applied to the depthwise kernel matrix (see constraints).
- **pointwise_constraint**: Constraint function applied to the pointwise kernel matrix (see constraints).
- **bias_constraint**: Constraint function applied to the bias vector (see constraints).

Input shape

4D tensor with shape:
`(batch, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, rows, cols, channels)` if data_format='channels_last'.

Output shape

4D tensor with shape:
`(batch, filters, new_rows, new_cols)` if data_format='channels_first'
or 4D tensor with shape:
`(batch, new_rows, new_cols, filters)` if data_format='channels_last'.
`rows` and `cols` values might have changed due to padding.

CLASS
alias of `SeparableConv2D`

class `conx.layers.`**`SimpleRNNLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

SimpleRNNLayer

Fully-connected RNN where the output is to be fed back to input.

Arguments

- **units**: Positive integer, dimensionality of the output space.
- **activation**: Activation function to use (see activations). If you pass None, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias**: Boolean, whether the layer uses a bias vector.
- **kernel_initializer**: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer**: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer**: Initializer for the bias vector (see initializers).
- **kernel_regularizer**: Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer**: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer**: Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer**: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint**: Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint**: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).

- **bias_constraint**: Constraint function applied to the bias vector (see constraints).
- **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

CLASS
alias of `SimpleRNN`

class `conx.layers.`**`SpatialDropout1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

SpatialDropout1DLayer

Spatial 1D version of Dropout.

This version performs the same function as Dropout, however it drops entire 1D feature maps instead of individual elements. If adjacent frames within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout1D will help promote independence between feature maps and should be used instead.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.

Input shape

3D tensor with shape:
`(samples, timesteps, channels)`

Output shape

Same as input

References

- Efficient Object Localization Using Convolutional Networks

CLASS
alias of `SpatialDropout1D`

class `conx.layers.`**`SpatialDropout2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

SpatialDropout2DLayer

Spatial 2D version of Dropout.

This version performs the same function as Dropout, however it drops entire 2D feature maps instead of individual elements. If adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout2D will help promote independence between feature maps and should be used instead.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.
- **data_format**: 'channels_first' or 'channels_last'. In 'channels_first' mode, the channels dimension (the depth) is at index 1, in 'channels_last' mode is it at index 3. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape:
`(samples, channels, rows, cols)` if data_format='channels_first'
or 4D tensor with shape:
`(samples, rows, cols, channels)` if data_format='channels_last'.

Output shape

Same as input

References

- Efficient Object Localization Using Convolutional Networks

CLASS
alias of `SpatialDropout2D`

class `conx.layers.`**`SpatialDropout3DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

SpatialDropout3DLayer

Spatial 3D version of Dropout.

This version performs the same function as Dropout, however it drops entire 3D feature maps instead of individual elements. If adjacent voxels within feature maps are strongly correlated (as is normally the case in early convolution layers) then regular dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, SpatialDropout3D will help promote independence between feature maps and should be used instead.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.
- **data_format**: 'channels_first' or 'channels_last'. In 'channels_first' mode, the channels dimension (the depth) is at index 1, in 'channels_last' mode is it at index 4. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape:
`(samples, channels, dim1, dim2, dim3)` if data_format='channels_first'
or 5D tensor with shape:
`(samples, dim1, dim2, dim3, channels)` if data_format='channels_last'.

Output shape

Same as input

References

- Efficient Object Localization Using Convolutional Networks

CLASS
alias of `SpatialDropout3D`

class `conx.layers.`**`ThresholdedReLULayer`**(*name*, **args*, ***params*)
Bases: *`conx.layers.BaseLayer`*

ThresholdedReLULayer

Thresholded Rectified Linear Unit.

It follows:
`f(x) = x for x > theta`,
`f(x) = 0 otherwise`.

Input shape

Arbitrary. Use the keyword argument `input_shape`
(tuple of integers, does not include the samples axis)
when using this layer as the first layer in a model.

Output shape

Same shape as the input.

Arguments

- **theta**: float >= 0. Threshold location of activation.

References

- Zero-Bias Autoencoders and the Benefits of Co-Adapting Features

CLASS
alias of ThresholdedReLU

class conx.layers.**TimeDistributedLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

TimeDistributedLayer

This wrapper applies a layer to every temporal slice of an input.

The input should be at least 3D, and the dimension of index one
will be considered to be the temporal dimension.

Consider a batch of 32 samples,
where each sample is a sequence of 10 vectors of 16 dimensions.
The batch input shape of the layer is then `(32, 10, 16)`,
and the `input_shape`, not including the samples dimension, is `(10, 16)`.

You can then use `TimeDistributed` to apply a `Dense` layer
to each of the 10 timesteps, independently:

```
# as the first layer in a model
model = Sequential()
model.add(TimeDistributed(Dense(8), input_shape=(10, 16)))
# now model.output_shape == (None, 10, 8)
```

The output will then have shape `(32, 10, 8)`.

In subsequent layers, there is no need for the `input_shape`:

```
model.add(TimeDistributed(Dense(32)))
# now model.output_shape == (None, 10, 32)
```

The output will then have shape `(32, 10, 32)`.

`TimeDistributed` can be used with arbitrary layers, not just `Dense`,
for instance with a `Conv2D` layer:

```
model = Sequential()
model.add(TimeDistributed(Conv2D(64, (3, 3)),
              input_shape=(10, 299, 299, 3)))
```

Arguments

- **layer**: a layer instance.

CLASS
alias of TimeDistributed

class conx.layers.**UpSampling1DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

UpSampling1DLayer

Upsampling layer for 1D inputs.

Repeats each temporal step `size` times along the time axis.

Arguments

- **size**: integer. Upsampling factor.

Input shape

3D tensor with shape: `(batch, steps, features)`.

Output shape

3D tensor with shape: `(batch, upsampled_steps, features)`.

CLASS
alias of `UpSampling1D`

class `conx.layers.`**`UpSampling2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

UpSampling2DLayer

Upsampling layer for 2D inputs.

Repeats the rows and columns of the data
by size[0] and size[1] respectively.

Arguments

- **size**: int, or tuple of 2 integers. The upsampling factors for rows and columns.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, upsampled_rows, upsampled_cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, upsampled_rows, upsampled_cols)`

CLASS
alias of `UpSampling2D`

class conx.layers.**UpSampling3DLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

UpSampling3DLayer

Upsampling layer for 3D inputs.

Repeats the 1st, 2nd and 3rd dimensions
of the data by size[0], size[1] and size[2] respectively.

Arguments

- **size**: int, or tuple of 3 integers. The upsampling factors for dim1, dim2 and dim3.
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, dim1, dim2, dim3, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, dim1, dim2, dim3)`

Output shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, upsampled_dim1, upsampled_dim2, upsampled_dim3, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, upsampled_dim1, upsampled_dim2, upsampled_dim3)`

CLASS
alias of `UpSampling3D`

class conx.layers.**WrapperLayer**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

WrapperLayer

Abstract wrapper base class.

Wrappers take another layer and augment it in various ways.
Do not use this class as a layer, it is only an abstract base class.
Two usable wrappers are the `TimeDistributed` and `Bidirectional` wrappers.

Arguments

- **layer**: The layer to be wrapped.

CLASS
alias of `Wrapper`

class `conx.layers.`**`ZeroPadding1DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ZeroPadding1DLayer

Zero-padding layer for 1D input (e.g. temporal sequence).

Arguments

- **padding**: int, or tuple of int (length 2), or dictionary.
  - If int: How many zeros to add at the beginning and end of the padding dimension (axis 1).
  - If tuple of int (length 2): How many zeros to add at the beginning and at the end of the padding dimension (`(left_pad, right_pad)`).

Input shape

3D tensor with shape `(batch, axis_to_pad, features)`

Output shape

3D tensor with shape `(batch, padded_axis, features)`

CLASS
alias of `ZeroPadding1D`

class `conx.layers.`**`ZeroPadding2DLayer`**(*name*, **args*, ***params*)
Bases: *conx.layers.BaseLayer*

ZeroPadding2DLayer

Zero-padding layer for 2D input (e.g. picture).

This layer can add rows and columns of zeros
at the top, bottom, left and right side of an image tensor.

Arguments

- **padding**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
  - If int: the same symmetric padding is applied to width and height.
  - If tuple of 2 ints: interpreted as two different symmetric padding values for height and width: `(symmetric_height_pad, symmetric_width_pad)`.
  - If tuple of 2 tuples of 2 ints: interpreted as `((top_pad, bottom_pad), (left_pad, right_pad))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, height, width, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, height, width)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, rows, cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, rows, cols)`

Output shape

4D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, padded_rows, padded_cols, channels)`
- If `data_format` is `"channels_first"`: `(batch, channels, padded_rows, padded_cols)`

CLASS

alias of `ZeroPadding2D`

class `conx.layers.`**`ZeroPadding3DLayer`**(*name*, **args*, ***params*)

Bases: *conx.layers.BaseLayer*

ZeroPadding3DLayer

Zero-padding layer for 3D data (spatial or spatio-temporal).

Arguments

- **padding**: int, or tuple of 2 ints, or tuple of 2 tuples of 2 ints.
  - If int: the same symmetric padding is applied to width and height.
  - If tuple of 2 ints: interpreted as two different symmetric padding values for height and width: `(symmetric_dim1_pad, symmetric_dim2_pad, symmetric_dim3_pad)`.
  - If tuple of 2 tuples of 2 ints: interpreted as `((left_dim1_pad, right_dim1_pad), (left_dim2_pad, right_dim2_pad), (left_dim3_pad, right_dim3_pad))`
- **data_format**: A string, one of `channels_last` (default) or `channels_first`. The ordering of the dimensions in the inputs. `channels_last` corresponds to inputs with shape `(batch, spatial_dim1, spatial_dim2, spatial_dim3, channels)` while `channels_first` corresponds to inputs with shape `(batch, channels, spatial_dim1, spatial_dim2, spatial_dim3)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Input shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad, depth)`
- If `data_format` is `"channels_first"`: `(batch, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)`

Output shape

5D tensor with shape:

- If `data_format` is `"channels_last"`: `(batch, first_padded_axis, second_padded_axis, third_axis_to_pad, depth)`
- If `data_format` is `"channels_first"`: `(batch, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)`

CLASS

alias of `ZeroPadding3D`

conx.layers.**process_class_docstring**(*docstring*)

conx.network module

The network module contains the code for the Network class.

class conx.network.**InterruptHandler**(*sig=<Signals.SIGINT: 2>*)
Bases: object

Class for handling interrupts so that state is not left in inconsistant situation.

class conx.network.**Network**(*name*, **sizes*, ***config*)
Bases: object

The main class for the conx neural network package.

OPTIMIZERS = ('sgd', 'rmsprop', 'adagrad', 'adadelta', 'adam', 'adamax', 'nadam', 'tfoptimizer')

add(*layer*)
Add a layer to the network layer connections. Order is not important, unless using the default net.connect() form.

build_svg(*opts={}*)
opts - temporary override of config

includes: "font_size": 12, "border_top": 25, "border_bottom": 25, "hspace": 100, "vspace": 50, "image_maxdim": 200

See .config for all options.

build_widget(*width='100%'*, *height='550px'*)
Build the control-panel for Jupyter widgets. Requires running in a notebook/jupyterlab.

compile(***kwargs*)
Check and compile the network.

connect(*from_layer_name=None*, *to_layer_name=None*)
Connect two layers together if called with arguments. If called with no arguments, then it will make a sequential run through the layers in order added.

describe_connection_to(*layer1*, *layer2*)
Returns a textual description of the weights for the SVG tooltip.

display_component(*vector*, *component*, ***opts*)
vector is a list, one each per output layer. component is "errors" or "targets"

get_input(*i*)
Get an input from the internal dataset and format it in the human API.

get_inputs_length()
Get the number of input patterns.

get_target(*i*)
Get a target from the internal dataset and format it in the human API.

get_targets_length()
Get the number of target patterns.

get_test_input(*i*)
Get a test input from the internal dataset and format it in the human API.

get_test_inputs_length()
Get the number of test input patterns.

get_test_target(*i*)
Get a test target from the internal dataset and format it in the human API.

get_test_targets_length()
Get the number of test target patterns.

get_train_input(*i*)
Get a training input from the internal dataset and format it in the human API.

get_train_inputs_length()
Get the number of training input patterns.

get_train_target(*i*)
Get a training target from the internal dataset and format it in the human API.

get_train_targets_length()
Get the number of training target patterns.

load(*filename=None*)
Load the weights from a file.

load_mnist_dataset(*verbose=True*)
Load the Keras MNIST dataset and format it as images.

pp(**args*, ***opts*)
Pretty-print a vector.

ppf(*vector*, ***opts*)
Pretty-format a vector.

propagate(*input*, *batch_size=32*)
Propagate an input (in human API) through the network. If visualizing, the network image will be updated.

propagate_from(*layer_name*, *input*, *output_layer_names=None*, *batch_size=32*)
Propagate activations from the given layer name to the output layers.

propagate_to(*layer_name*, *inputs*, *batch_size=32*, *visualize=True*)
Computes activation at a layer. Side-effect: updates visualized SVG.

propagate_to_image(*layer_name*, *input*, *batch_size=32*)
Gets an image of activations at a layer.

rescale_inputs(*old_range*, *new_range*, *new_dtype*)
Rescale the inputs. WIP.

reset()
Reset all of the weights/biases in a network. The magnitude is based on the size of the network.

reshape_inputs(*new_shape*, *verbose=True*)
Reshape the input vectors. WIP.

retrain(***overrides*)
Call network.train() again with same options as last call, unless overrides.

save(*filename=None*)
Save the weights to a file.

set_dataset(*pairs*, *verbose=True*)
Set the human-specified dataset to a proper keras dataset.

Multi-inputs or multi-outputs must be: [vector, vector, ...] for each layer input/target pairing.

set_dataset_direct(*inputs*, *targets*, *verbose=True*)
Set the inputs/targets in the specific internal format:

[input-vector, input-vector, ...] if single input layer

[[input-layer-1-vectors ...], [input-layer-2-vectors ...], ...] if input target layers

[target-vector, target-vector, ...] if single output layer

[[target-layer-1-vectors], [target-layer-2-vectors], ...] if multi target layers

set_input_layer_order(**layer_names*)
When multiple input banks, you must set this.

set_output_layer_order(**layer_names*)
When multiple output banks, you must set this.

set_targets_to_categories(*num_classes*)
Given net.labels are integers, set the net.targets to one_hot() categories.

shuffle_dataset(*verbose=True*)
Shuffle the inputs/targets. WIP.

slice_dataset(*start=None*, *stop=None*, *verbose=True*)
Cut out some input/targets.

net.slice_dataset(100) - reduce to first 100 inputs/targets net.slice_dataset(100, 200) - reduce to second 100 inputs/targets

split_dataset(*split=0.5*, *verbose=True*)
Split the inputs/targets into training/test datasets.

summary()
Print out a summary of the network.

summary_dataset()
Print out a summary of the dataset.

test(*inputs=None*, *targets=None*, *batch_size=32*, *tolerance=0.1*)
Requires items in proper internal format, if given (for now).

train(*epochs=1*, *accuracy=None*, *batch_size=None*, *report_rate=1*, *tolerance=0.1*, *verbose=1*, *shuffle=True*, *class_weight=None*, *sample_weight=None*)
Train the network.

train_one(*inputs*, *targets*, *batch_size=32*)
Train on one input/target pair. Requires internal format.

conx.utils module

conx.utils.**autoname**(*index*, *sizes*)
Given an index and list of sizes, return a name for the layer.

conx.utils.**one_hot**(*vector*, *categories*)
Given a vector of integers (i.e. labels), return a numpy array of one-hot vectors.

conx.utils.**rescale_numpy_array**(*a*, *old_range*, *new_range*, *new_dtype*)
Given a vector, old min/max, a new min/max and a numpy type, create a new vector scaling the old values.

conx.utils.**topological_sort**(*net*, *layers*)
Given a conx network and list of layers, produce a topological sorted list, from input(s) to output(s).

conx.utils.**valid_shape**(*x*)
Is this a valid shape for Keras layers?

conx.utils.**valid_vshape**(*x*)
Is this a valid shape (i.e., size) to display vectors using PIL?

conx.utils.**visit**(*layer*, *stack*)
Utility function for topological_sort.

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

D

E

S

T

U

V

W

Z