Getting started with the Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_dim=784),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

Specifying the input shape

The model needs to know what input shape it should expect. For this reason, the first layer in a `Sequential` model (and only the first, because following layers can do automatic shape inference) needs to receive information about its input shape. There are several possible ways to do this:

- pass an `input_shape` argument to the first layer. This is a shape tuple (a tuple of integers or `None` entries, where `None` indicates that any positive integer may be expected). In `input_shape`, the batch dimension is not included.
- pass instead a `batch_input_shape` argument, where the batch dimension is included. This is useful for specifying a fixed batch size (e.g. with stateful RNNs).
- some 2D layers, such as `Dense`, support the specification of their input shape via the argument `input_dim`, and some 3D temporal layers support the arguments `input_dim` and `input_length`.

As such, the following three snippets are strictly equivalent:

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, batch_input_shape=(None, 784)))
# note that batch dimension is "None" here,
# so the model will be able to process batches of any size.
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

And so are the following three snippets:

```
model = Sequential()
model.add(LSTM(32, input_shape=(10, 64)))
```

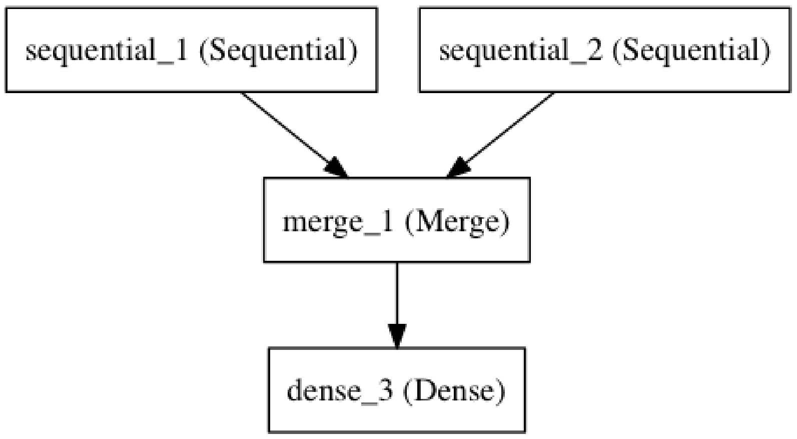
```
model = Sequential()
model.add(LSTM(32, batch_input_shape=(None, 10, 64)))
```

```
model = Sequential()
model.add(LSTM(32, input_length=10, input_dim=64))
```

The Merge layer

Multiple `Sequential` instances can be merged into a single output via a `Merge` layer. The output is a layer that can be added as first layer in a new `Sequential` model. For instance, here's a model with two separate input branches getting merged:

```
from keras.layers import Merge

left_branch = Sequential()
left_branch.add(Dense(32, input_dim=784))

right_branch = Sequential()
right_branch.add(Dense(32, input_dim=784))

merged = Merge([left_branch, right_branch], mode='concat')

final_model = Sequential()
final_model.add(merged)
final_model.add(Dense(10, activation='softmax'))
```



Such a two-branch model can then be trained via e.g.:

```
final_model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
final_model.fit([input_data_1, input_data_2], targets)  # we pass one data array per model inpu
```

The `Merge` layer supports a number of pre-defined modes:

- `sum` (default): element-wise sum
- `concat`: tensor concatenation. You can specify the concatenation axis via the argument `concat_axis`.
- `mul`: element-wise multiplication
- `ave`: tensor average
- `dot`: dot product. You can specify which axes to reduce along via the argument `dot_axes`.
- `cos`: cosine proximity between vectors in 2D tensors.

You can also pass a function as the `mode` argument, allowing for arbitrary transformations:

```
merged = Merge([left_branch, right_branch], mode=lambda x, y: x - y)
```

Now you know enough to be able to define *almost* any model with Keras. For complex models that cannot be expressed via `Sequential` and `Merge`, you can use the functional API.

Compilation

Before training a model, you need to configure the learning process, which is done via the `compile` method. It receives three arguments:

- an optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class. See: optimizers.
- a loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as `categorical_crossentropy` or `mse`), or it can be an objective function. See: objectives.
- a list of metrics. For any classification problem you will want to set this to `metrics=['accuracy']`. A metric could be the string identifier of an existing metric (only `accuracy` is supported at this point), or a custom metric function.

```
# for a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# for a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# for a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, you will typically use the `fit` function. Read its documentation here.

```
# for a single-input model with 2 classes (binary):

model = Sequential()
model.add(Dense(1, input_dim=784, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# generate dummy data
import numpy as np
data = np.random.random((1000, 784))
labels = np.random.randint(2, size=(1000, 1))

# train the model, iterating on the data in batches
# of 32 samples
model.fit(data, labels, nb_epoch=10, batch_size=32)
```

```
# for a multi-input model with 10 classes:

left_branch = Sequential()
left_branch.add(Dense(32, input_dim=784))

right_branch = Sequential()
right_branch.add(Dense(32, input_dim=784))

merged = Merge([left_branch, right_branch], mode='concat')

model = Sequential()
model.add(merged)
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# generate dummy data
import numpy as np
from keras.utils.np_utils import to_categorical
data_1 = np.random.random((1000, 784))
data_2 = np.random.random((1000, 784))

# these are integers between 0 and 9
labels = np.random.randint(10, size=(1000, 1))
# we convert the labels to a binary matrix of size (1000, 10)
# for use with categorical_crossentropy
labels = to_categorical(labels, 10)

# train the model
# note that we are passing a list of Numpy arrays as training data
# since the model has 2 inputs
model.fit([data_1, data_2], labels, nb_epoch=10, batch_size=32)
```

Examples

Here are a few examples to get you started!

In the examples folder, you will also find example models for real datasets:

- CIFAR10 small images classification: Convolutional Neural Network (CNN) with realtime data augmentation
- IMDB movie review sentiment classification: LSTM over sequences of words
- Reuters newswires topic classification: Multilayer Perceptron (MLP)
- MNIST handwritten digits classification: MLP & CNN
- Character-level text generation with LSTM

...and more.

Multilayer Perceptron (MLP) for multi-class softmax classification:

```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
          batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

Alternative implementation of a similar MLP:

```python
model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

MLP for binary classification:

```python
model = Sequential()
model.add(Dense(64, input_dim=20, init='uniform', activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

VGG-like convnet:

```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import SGD

model = Sequential()
# input: 100x100 images with 3 channels -> (3, 100, 100) tensors.
# this applies 32 convolution filters of size 3x3 each.
model.add(Convolution2D(32, 3, 3, border_mode='valid', input_shape=(3, 100, 100)))
model.add(Activation('relu'))
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(64, 3, 3, border_mode='valid'))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
# Note: Keras does automatic shape inference.
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(10))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(X_train, Y_train, batch_size=32, nb_epoch=1)
```

Sequence classification with LSTM:

```python
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 256, input_length=maxlen))
model.add(LSTM(output_dim=128, activation='sigmoid', inner_activation='hard_sigmoid'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(X_train, Y_train, batch_size=16, nb_epoch=10)
score = model.evaluate(X_test, Y_test, batch_size=16)
```

Architecture for learning image captions with a convnet and a Gated Recurrent Unit:

(word-level embedding, caption of maximum length 16 words).

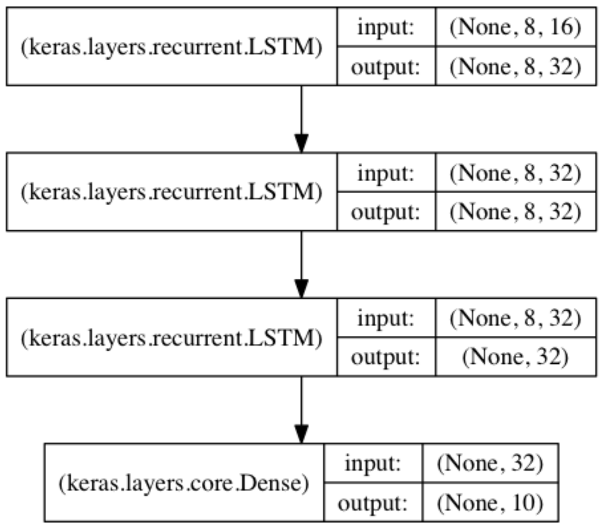
Note that getting this to work well will require using a bigger convnet, initialized with pre-trained weights.

```
max_caption_len = 16
vocab_size = 10000

# first, let's define an image model that
# will encode pictures into 128-dimensional vectors.
# it should be initialized with pre-trained weights.
image_model = Sequential()
image_model.add(Convolution2D(32, 3, 3, border_mode='valid', input_shape=(3, 100, 100)))
image_model.add(Activation('relu'))
image_model.add(Convolution2D(32, 3, 3))
image_model.add(Activation('relu'))
image_model.add(MaxPooling2D(pool_size=(2, 2)))

image_model.add(Convolution2D(64, 3, 3, border_mode='valid'))
image_model.add(Activation('relu'))
image_model.add(Convolution2D(64, 3, 3))
image_model.add(Activation('relu'))
image_model.add(MaxPooling2D(pool_size=(2, 2)))

image_model.add(Flatten())
image_model.add(Dense(128))

# let's load the weights from a save file.
image_model.load_weights('weight_file.h5')

# next, let's define a RNN model that encodes sequences of words
# into sequences of 128-dimensional word vectors.
language_model = Sequential()
language_model.add(Embedding(vocab_size, 256, input_length=max_caption_len))
language_model.add(GRU(output_dim=128, return_sequences=True))
language_model.add(TimeDistributed(Dense(128)))

# let's repeat the image vector to turn it into a sequence.
image_model.add(RepeatVector(max_caption_len))

# the output of both models will be tensors of shape (samples, max_caption_len, 128).
# let's concatenate these 2 vector sequences.
model = Sequential()
model.add(Merge([image_model, language_model], mode='concat', concat_axis=-1))
# let's encode this vector sequence into a single vector
model.add(GRU(256, return_sequences=False))
# which will be used to compute a probability
# distribution over what the next word in the caption should be!
model.add(Dense(vocab_size))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

# "images" is a numpy float array of shape (nb_samples, nb_channels=3, width, height).
# "captions" is a numpy integer array of shape (nb_samples, max_caption_len)
# containing word index sequences representing partial captions.
# "next_words" is a numpy float array of shape (nb_samples, vocab_size)
# containing a categorical encoding (0s and 1s) of the next word in the corresponding
# partial caption.
model.fit([images, partial_captions], next_words, batch_size=16, nb_epoch=100)
```

Stacked LSTM for sequence classification

In this model, we stack 3 LSTM layers on top of each other, making the model capable of learning higher-level temporal representations.

The first two LSTMs return their full output sequences, but the last one only returns the last step in its output sequence, thus dropping the temporal dimension (i.e. converting the input sequence into a single vector).



```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
nb_classes = 10

# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a sequence of vectors of dimensio
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))

# generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))

model.fit(x_train, y_train,
          batch_size=64, nb_epoch=5,
          validation_data=(x_val, y_val))
```

Same stacked LSTM model, rendered "stateful"

A stateful recurrent model is one for which the internal states (memories) obtained after processing a batch of samples are reused as initial states for the samples of the next batch. This allows to process longer sequences while keeping computational complexity manageable.

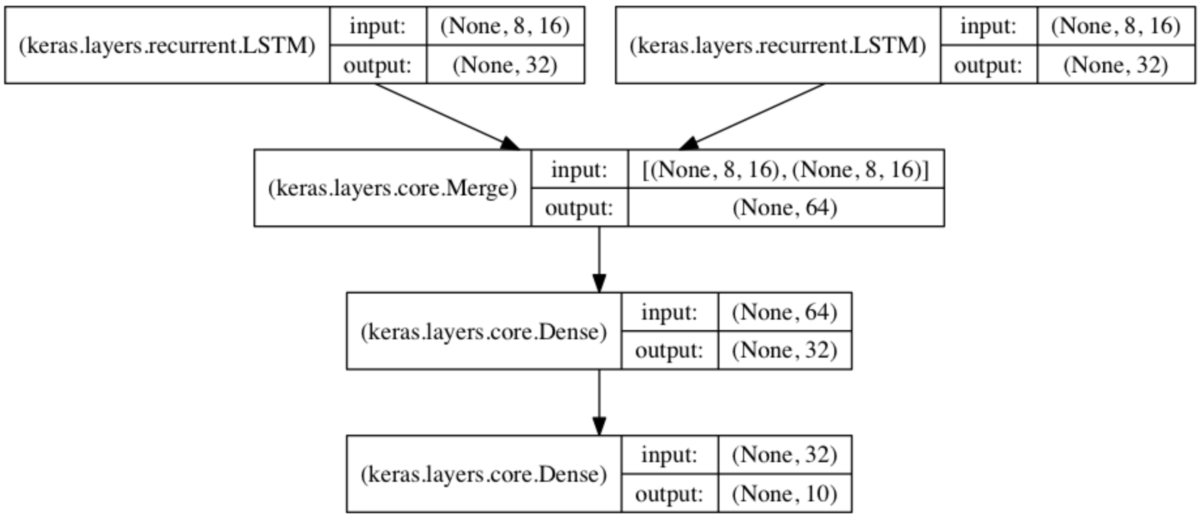
You can read more about stateful RNNs in the FAQ.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
nb_classes = 10
batch_size = 32

# expected input batch shape: (batch_size, timesteps, data_dim)
# note that we have to provide the full batch_input_shape since the network is stateful.
# the sample of index i in batch k is the follow-up for the sample i in batch k-1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, nb_classes))

# generate dummy validation data
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, nb_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, nb_epoch=5,
          validation_data=(x_val, y_val))
```

Two merged LSTM encoders for classification over two parallel sequences

In this model, two input sequences are encoded into vectors by two separate LSTM modules.

These two vectors are then concatenated, and a fully connected network is trained on top of the concatenated representations.



(keras.layers.recurrent.LSTM)
input: (None, 8, 16)
output: (None, 32)
(keras.layers.recurrent.LSTM)
input: (None, 8, 16)
output: (None, 32)
(keras.layers.core.Merge)
input: [(None, 8, 16), (None, 8, 16)]
output: (None, 64)
(keras.layers.core.Dense)
input: (None, 64)
output: (None, 32)
(keras.layers.core.Dense)
input: (None, 32)
output: (None, 10)

```python
from keras.models import Sequential
from keras.layers import Merge, LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
nb_classes = 10

encoder_a = Sequential()
encoder_a.add(LSTM(32, input_shape=(timesteps, data_dim)))

encoder_b = Sequential()
encoder_b.add(LSTM(32, input_shape=(timesteps, data_dim)))

decoder = Sequential()
decoder.add(Merge([encoder_a, encoder_b], mode='concat'))
decoder.add(Dense(32, activation='relu'))
decoder.add(Dense(nb_classes, activation='softmax'))

decoder.compile(loss='categorical_crossentropy',
                optimizer='rmsprop',
                metrics=['accuracy'])

# generate dummy training data
x_train_a = np.random.random((1000, timesteps, data_dim))
x_train_b = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))

# generate dummy validation data
x_val_a = np.random.random((100, timesteps, data_dim))
x_val_b = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))

decoder.fit([x_train_a, x_train_b], y_train,
            batch_size=64, nb_epoch=5,
            validation_data=([x_val_a, x_val_b], y_val))
```

Getting started with the Keras functional API

The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

This guide assumes that you are already familiar with the `Sequential` model.

Let's start with something simple.

First example: fully connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

- A layer instance is callable (on a tensor), and it returns a tensor
- Input tensor(s) and output tensor(s) can then be used to define a `Model`
- Such a model can be trained just like Keras `Sequential` models.

```
from keras.layers import Input, Dense
from keras.models import Model

# this returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# this creates a model that includes
# the Input layer and three Dense layers
model = Model(input=inputs, output=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

All models are callable, just like layers

With the functional API, it is easy to re-use trained models: you can treat any model as if it were a layer, by calling it on a tensor. Note that by calling a model you aren't just re-using the *architecture* of the model, you are also re-using its weights.

```
x = Input(shape=(784,))
# this works, and returns the 10-way softmax we defined above.
y = model(x)
```

This can allow, for instance, to quickly create models that can process *sequences* of inputs. You could turn an image classification model into a video classification model, in just one line.

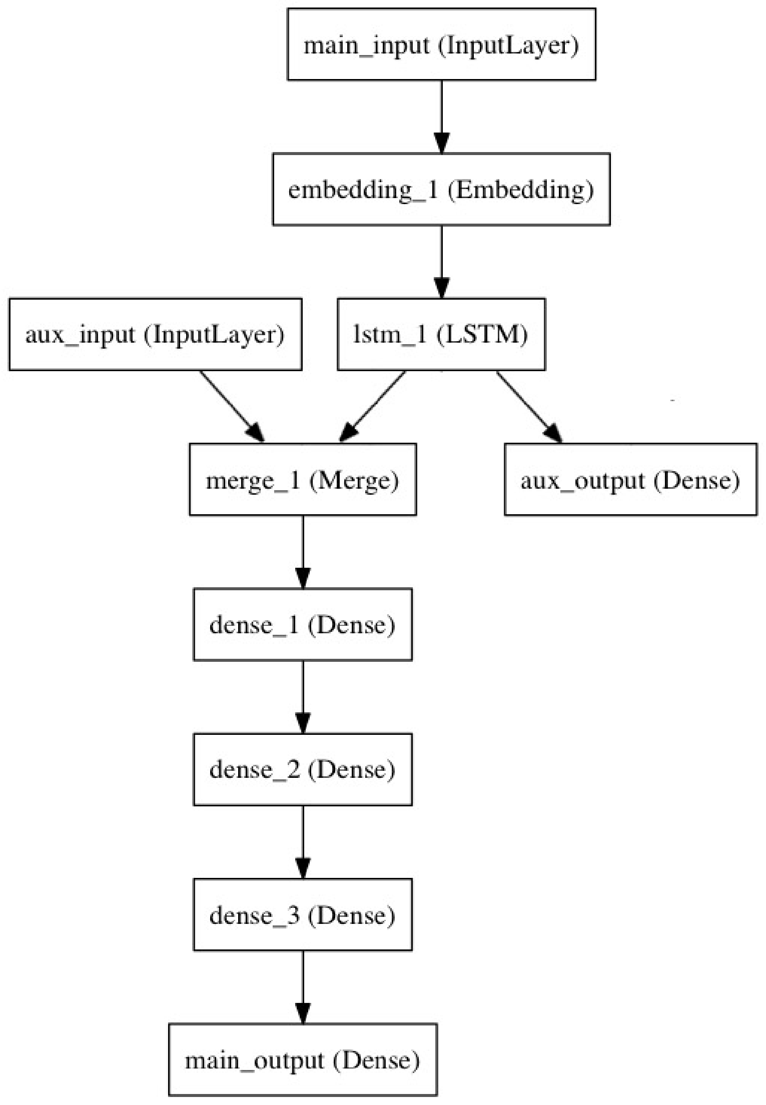
```
from keras.layers import TimeDistributed

# input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

# this applies our previous model to every timestep in the input sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

Multi-input and multi-output models

Here's a good use case for the functional API: models with multiple inputs and outputs. The functional API makes it easy to manipulate a large number of intertwined datastreams.

Let's consider the following model. We seek to predict how many retweets and likes a news headline will receive on Twitter. The main input to the model will be the headline itself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the time of day when the headline was posted, etc. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Here's what our model looks like:



Let's implement it with the functional API.

The main input will receive the headline, as a sequence of integers (each integer encodes a word). The integers will be between 1 and 10,000 (a vocabulary of 10,000 words) and the sequences will be 100 words long.

```
from keras.layers import Input, Embedding, LSTM, Dense, merge
from keras.models import Model

# headline input: meant to receive sequences of 100 integers, between 1 and 10000.
# note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# this embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# a LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

Here we insert the auxiliary loss, allowing the LSTM and Embedding layer to be trained smoothly even though the main loss will be much higher in the model.

```
auxiliary_loss = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

At this point, we feed into the model our auxiliary input data by concatenating it with the LSTM output:

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = merge([lstm_out, auxiliary_input], mode='concat')

# we stack a deep fully-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# and finally we add the main logistic regression layer
main_loss = Dense(1, activation='sigmoid', name='main_output')(x)
```

This defines a model with two inputs and two outputs:

```
model = Model(input=[main_input, auxiliary_input], output=[main_loss, auxiliary_loss])
```

We compile the model and assign a weight of 0.2 to the auxiliary loss. To specify different `loss_weights` or `loss` for each different output, you can use a list or a dictionary. Here we pass a single loss as the `loss` argument, so the same loss will be used on all outputs.

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

We can train the model by passing it lists of input arrays and target arrays:

```
model.fit([headline_data, additional_data], [labels, labels],
          nb_epoch=50, batch_size=32)
```

Since our inputs and outputs are named (we passed them a "name" argument), We could also have compiled the model via:

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy'},
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# and trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          nb_epoch=50, batch_size=32)
```

Shared layers

Another good use for the functional API are models that use shared layers. Let's take a look at shared layers.

Let's consider a dataset of tweets. We want to build a model that can tell whether two tweets are from the same person or not (this can allow us to compare users by the similarity of their tweets, for instance).

One way to achieve this is to build a model that encodes two tweets into two vectors, concatenates the vectors and adds a logistic regression of top, outputting a probability that the two tweets share the same author. The model would then be trained on positive tweet pairs and negative tweet pairs.

Because the problem is symmetric, the mechanism that encodes the first tweet should be reused (weights and all) to encode the second tweet. Here we use a shared LSTM layer to encode the tweets.

Let's build this with the functional API. We will take as input for a tweet a binary matrix of shape `(140, 256)`, i.e. a sequence of 140 vectors of size 256, where each dimension in the 256-dimensional vector encodes the presence/absence of a character (out of an alphabet of 256 frequent characters).

```
from keras.layers import Input, LSTM, Dense, merge
from keras.models import Model

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

To share a layer across different inputs, simply instantiate the layer once, then call it on as many inputs as you want:

```
# this layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# when we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# we can then concatenate the two vectors:
merged_vector = merge([encoded_a, encoded_b], mode='concat', concat_axis=-1)

# and add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# we define a trainable model linking the
# tweet inputs to the predictions
model = Model(input=[tweet_a, tweet_b], output=predictions)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit([data_a, data_b], labels, nb_epoch=10)
```

Let's pause to take a look at how to read the shared layer's output or output shape.

The concept of layer "node"

Whenever you are calling a layer on some input, you are creating a new tensor (the output of the layer), and you are adding a "node" to the layer, linking the input tensor to the output tensor. When you are calling the same layer multiple times, that layer owns multiple nodes indexed as 0, 1, 2...

In previous versions of Keras, you could obtain the output tensor of a layer instance via `layer.get_output()`, or its output shape via `layer.output_shape`. You still can (except `get_output()` has been replaced by the property `output`). But what if a layer is connected to multiple inputs?

As long as a layer is only connected to one input, there is no confusion, and `.output` will return the one output of the layer:

```
a = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a
```

Not so if the layer has multiple inputs:

```
a = Input(shape=(140, 256))
b = Input(shape=(140, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output
```

```
>> AssertionError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.
```

Okay then. The following works:

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```

Simple enough, right?

The same is true for the properties `input_shape` and `output_shape`: as long as the layer has only one node, or as long as all nodes have the same input/output shape, then the notion of "layer output/input shape" is well defined, and that one shape will be returned by `layer.output_shape`/`layer.input_shape`. But if, for instance, you apply a same `Convolution2D` layer to an input of shape `(3, 32, 32)`, and then to an input of shape `(3, 64, 64)`, the layer will have multiple input/output shapes, and you will have to fetch them by specifying the index of the node they belong to:

```
a = Input(shape=(3, 32, 32))
b = Input(shape=(3, 64, 64))

conv = Convolution2D(16, 3, 3, border_mode='same')
conved_a = conv(a)

# only one input so far, the following will work:
assert conv.input_shape == (None, 3, 32, 32)

conved_b = conv(b)
# now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 3, 32, 32)
assert conv.get_input_shape_at(1) == (None, 3, 64, 64)
```

More examples

Code examples are still the best way to get started, so here are a few more.

Inception module

For more information about the Inception architecture, see Going Deeper with Convolutions.

```
from keras.layers import merge, Convolution2D, MaxPooling2D, Input

input_img = Input(shape=(3, 256, 256))

tower_1 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(input_img)
tower_1 = Convolution2D(64, 3, 3, border_mode='same', activation='relu')(tower_1)

tower_2 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(input_img)
tower_2 = Convolution2D(64, 5, 5, border_mode='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), border_mode='same')(input_img)
tower_3 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(tower_3)

output = merge([tower_1, tower_2, tower_3], mode='concat', concat_axis=1)
```

Residual connection on a convolution layer

For more information about residual networks, see Deep Residual Learning for Image Recognition.

```
from keras.layers import merge, Convolution2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(3, 256, 256))
# 3x3 conv with 3 output channels (same as input channels)
y = Convolution2D(3, 3, 3, border_mode='same')(x)
# this returns x + y.
z = merge([x, y], mode='sum')
```

Shared vision model

This model re-uses the same image-processing module on two inputs, to classify whether two MNIST digits are the same digit or different digits.

```
from keras.layers import merge, Convolution2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# first, define the vision modules
digit_input = Input(shape=(1, 27, 27))
x = Convolution2D(64, 3, 3)(digit_input)
x = Convolution2D(64, 3, 3)(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# then define the tell-digits-apart model
digit_a = Input(shape=(1, 27, 27))
digit_b = Input(shape=(1, 27, 27))

# the vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = merge([out_a, out_b], mode='concat')
out = Dense(1, activation='sigmoid')(concatenated)

classification_model = Model([digit_a, digit_b], out)
```

Visual question answering model

This model can select the correct one-word answer when asked a natural-language question about a picture.

It works by encoding the question into a vector, encoding the image into a vector, concatenating the two, and training on top a logistic regression over some vocabulary of potential answers.

```
from keras.layers import Convolution2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense, merge
from keras.models import Model, Sequential

# first, let's define a vision model using a Sequential model.
# this model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same', input_shape=(3,
vision_model.add(Convolution2D(64, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(128, 3, 3, activation='relu', border_mode='same'))
vision_model.add(Convolution2D(128, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(256, 3, 3, activation='relu', border_mode='same'))
vision_model.add(Convolution2D(256, 3, 3, activation='relu'))
vision_model.add(Convolution2D(256, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# now let's get a tensor with the output of our vision model:
image_input = Input(shape=(3, 224, 224))
encoded_image = vision_model(image_input)

# next, let's define a language model to encode the question into a vector.
# each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)(question_input
encoded_question = LSTM(256)(embedded_question)

# let's concatenate the question vector and the image vector:
merged = merge([encoded_question, encoded_image], mode='concat')

# and let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# this is our final model:
vqa_model = Model(input=[image_input, question_input], output=output)

# the next stage would be training this model on actual data.
```

Video question answering model

Now that we have trained our image QA model, we can quickly turn it into a video QA model. With appropriate training, you will be able to show it a short video (e.g. 100-frame human action) and ask a natural language question about the video (e.g. "what sport is the boy playing?" -> "football").

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 3, 224, 224))
# this is our video encoded via the previously trained vision_model (weights are reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input)  # the output will be a seq
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output will be a vector

# this is a model-level representation of the question encoder, reusing the same weights as bef
question_encoder = Model(input=question_input, output=encoded_question)

# let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# and this is our video question answering model:
merged = merge([encoded_video, encoded_video_question], mode='concat')
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(input=[video_input, video_question_input], output=output)
```

Keras FAQ: Frequently Asked Keras Questions

How should I cite Keras?

Please cite Keras in your publications if it helps your research. Here is an example BibTeX entry:

```
@misc{chollet2015keras,
  title={Keras},
  author={Chollet, Fran\c{c}ois},
  year={2015},
  publisher={GitHub},
  howpublished={\url{https://github.com/fchollet/keras}},
}
```

How can I run Keras on GPU?

If you are running on the TensorFlow backend, your code will automatically run on GPU if any available GPU is detected. If you are running on the Theano backend, you can use one of the following methods:

Method 1: use Theano flags.

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

The name 'gpu' might have to be changed depending on your device's identifier (e.g. `gpu0`, `gpu1`, etc).

Method 2: set up your `.theanorc`: Instructions

Method 3: manually set `theano.config.device`, `theano.config.floatX` at the beginning of your code:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

How can I save a Keras model?

It is not recommended to use pickle or cPickle to save a Keras model.

You can use `model.save(filepath)` to save a Keras model into a single HDF5 file which will contain:

- the architecture of the model, allowing to re-create the model
- the weights of the model
- the training configuration (loss, optimizer)
- the state of the optimizer, allowing to resume training exactly where you left off.

You can then use `keras.models.load_model(filepath)` to reinstantiate your model. `load_model` will also take care of compiling the model using the saved training configuration (unless the model was never compiled in the first place).

Example:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

If you only need to save the **architecture of a model**, and not its weights or its training configuration, you can do:

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

The generated JSON / YAML files are human-readable and can be manually edited if needed.

You can then build a fresh model from this data:

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
model = model_from_yaml(yaml_string)
```

If you need to save the **weights of a model**, you can do so in HDF5 with the code below.

Note that you will first need to install HDF5 and the Python library h5py, which do not come bundled with Keras.

```
model.save_weights('my_model_weights.h5')
```

Assuming you have code for instantiating your model, you can then load the weights you saved into a model with the same architecture:

```
model.load_weights('my_model_weights.h5')
```

Why is the training loss much higher than the testing loss?

A Keras model has two modes: training and testing. Regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at testing time.

Besides, the training loss is the average of the losses over each batch of training data. Because your model is changing over time, the loss over the first batches of an epoch is generally higher than over the last batches. On the other hand, the testing loss for an epoch is computed using the model as it is at the end of the epoch, resulting in a lower loss.

How can I visualize the output of an intermediate layer?

You can build a Keras function that will return the output of a certain layer given a certain input, for example:

```
from keras import backend as K

# with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([X])[0]
```

Similarly, you could build a Theano and TensorFlow function directly.

Note that if your model has a different behavior in training and testing phase (e.g. if it uses `Dropout`, `BatchNormalization`, etc.), you will need to pass the learning phase flag to your function:

```
get_3rd_layer_output = K.function([model.layers[0].input, K.learning_phase()],
                                  [model.layers[3].output])

# output in test mode = 0
layer_output = get_3rd_layer_output([X, 0])[0]

# output in train mode = 1
layer_output = get_3rd_layer_output([X, 1])[0]
```

Another more flexible way of getting output from intermediate layers is to use the functional API. For example, if you have created an autoencoder for MNIST:

```
inputs = Input(shape=(784,))
encoded = Dense(32, activation='relu')(inputs)
decoded = Dense(784)(encoded)
model = Model(input=inputs, output=decoded)
```

After compiling and training the model, you can get the output of the data from the encoder like this:

```
encoder = Model(input=inputs, output=encoded)
X_encoded = encoder.predict(X)
```

How can I use Keras with datasets that don't fit in memory?

You can do batch training using `model.train_on_batch(X, y)` and `model.test_on_batch(X, y)`. See the models documentation.

Alternatively, you can write a generator that yields batches of training data and use the method `model.fit_generator(data_generator, samples_per_epoch, nb_epoch)`.

You can see batch training in action in our CIFAR10 example.

How can I interrupt training when the validation loss isn't decreasing anymore?

You can use an `EarlyStopping` callback:

```
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
model.fit(X, y, validation_split=0.2, callbacks=[early_stopping])
```

Find out more in the callbacks documentation.

How is the validation split computed?

If you set the `validation_split` argument in `model.fit` to e.g. 0.1, then the validation data used will be the *last 10%* of the data. If you set it to 0.25, it will be the last 25% of the data, etc.

Is the data shuffled during training?

Yes, if the `shuffle` argument in `model.fit` is set to `True` (which is the default), the training data will be randomly shuffled at each epoch.

Validation data is never shuffled.

How can I record the training / validation loss / accuracy at each epoch?

The `model.fit` method returns an `History` callback, which has a `history` attribute containing the lists of successive losses and other metrics.

```
hist = model.fit(X, y, validation_split=0.2)
print(hist.history)
```

How can I "freeze" Keras layers?

To "freeze" a layer means to exclude it from training, i.e. its weights will never be updated. This is useful in the context of fine-tuning a model, or using fixed embeddings for a text input.

You can pass a `trainable` argument (boolean) to a layer constructor to set a layer to be non-trainable:

```
frozen_layer = Dense(32, trainable=False)
```

Additionally, you can set the `trainable` property of a layer to `True` or `False` after instantiation. For this to take effect, you will need to call `compile()` on your model after modifying the `trainable` property. Here's an example:

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)

frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated during training
# (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

How can I use stateful RNNs?

Making a RNN stateful means that the states for the samples of each batch will be reused as initial states for the samples in the next batch.

When using stateful RNNs, it is therefore assumed that:

- all batches have the same number of samples
- If `X1` and `X2` are successive batches of samples, then `X2[i]` is the follow-up sequence to `X1[i]`, for every `i`.

To use statefulness in RNNs, you need to:

- explicitly specify the batch size you are using, by passing a `batch_input_shape` argument to the first layer in your model. It should be a tuple of integers, e.g. `(32, 10, 16)` for a 32-samples batch of sequences of 10 timesteps with 16 features per timestep.
- set `stateful=True` in your RNN layer(s).

To reset the states accumulated:

- use `model.reset_states()` to reset the states of all layers in the model
- use `layer.reset_states()` to reset the states of a specific stateful RNN layer

Example:

```
X  # this is our input data, of shape (32, 21, 16)
# we will feed it to our model in sequences of length 10

model = Sequential()
model.add(LSTM(32, batch_input_shape=(32, 10, 16), stateful=True))
model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(X[:, :10, :], np.reshape(X[:, 10, :], (32, 16)))

# the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(X[:, 10:20, :], np.reshape(X[:, 20, :], (32, 16)))

# let's reset the states of the LSTM layer:
model.reset_states()

# another way to do it in this case:
model.layers[0].reset_states()
```

Notes that the methods `predict`, `fit`, `train_on_batch`, `predict_classes`, etc. will *all* update the states of the stateful layers in a model. This allows you to do not only stateful training, but also stateful prediction.

How can I remove a layer from a Sequential model?

You can remove the last added layer in a Sequential model by calling `.pop()`:

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))

print(len(model.layers))  # "2"

model.pop()
print(len(model.layers))  # "1"
```

How can I use pre-trained models in Keras?

Code and pre-trained weights are available for the following image classification models:

- VGG-16
- VGG-19
- AlexNet

For an example of how to use such a pre-trained model for feature extraction or for fine-tuning, see this blog post.

The VGG-16 model is also the basis for several Keras example scripts:

- Style transfer
- Feature visualization
- Deep dream

About Keras models

There are two types of models available in Keras: the Sequential model and the Model class used with functional API.

These models have a number of methods in common:

- `model.summary()`: prints a summary representation of your model.
- `model.get_config()`: returns a dictionary containing the configuration of the model. The model can be reinstantiated from its config via:

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential:
model = Sequential.from_config(config)
```

- `model.get_weights()`: returns a list of all weight tensors in the model, as Numpy arrays.
- `model.set_weights(weights)`: sets the values of the weights of the model, from a list of Numpy arrays. The arrays in the list should have the same shape as those returned by `get_weights()`.
- `model.to_json()`: returns a representation of the model as a JSON string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the JSON string via:

```
from models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

- `model.to_yaml()`: returns a representation of the model as a YAML string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the YAML string via:

```
from models import model_from_yaml

yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

- `model.save_weights(filepath)`: saves the weights of the model as a HDF5 file.
- `model.load_weights(filepath)`: loads the weights of the model from a HDF5 file (created by `save_weights`).

The Sequential model API

To get started, read this guide to the Keras Sequential model.

Useful attributes of Model

- `model.layers` is a list of the layers added to the model.

Sequential model methods

compile

```
compile(self, optimizer, loss, metrics=[], sample_weight_mode=None)
```

Configures the learning process.

Arguments

- **optimizer**: str (name of optimizer) or optimizer object. See optimizers.
- **loss**: str (name of objective function) or objective function. See objectives.
- **metrics**: list of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`.
- **sample_weight_mode**: if you need to do timestep-wise sample weighting (2D weights), set this to "temporal". "None" defaults to sample-wise weights (1D).
- **kwargs**: for Theano backend, these are passed into K.function. Ignored for Tensorflow backend.

Example

```
    model = Sequential()
    model.add(Dense(32, input_shape=(500,)))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer='rmsprop',
          loss='categorical_crossentropy',
          metrics=['accuracy'])
```

fit

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[], validation_split=0.0, vali
```

Trains the model for a fixed number of epochs.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **batch_size**: integer. Number of samples per gradient update.
- **nb_epoch**: integer, the number of epochs to train the model.
- **verbose**: 0 for no logging to stdout, 1 for progress bar logging, 2 for one log line per epoch.
- **callbacks**: list of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: float (0. < x < 1). Fraction of the data to use as held-out validation data.
- **validation_data**: tuple (X, y) to be used as held-out validation data. Will override validation_split.
- **shuffle**: boolean or str (for 'batch'). Whether to shuffle the samples at each epoch. 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks.
- **class_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- **sample_weight**: Numpy array of weights for the training samples, used for scaling the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples
    - (**1**:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

Computes the loss on some input data, batch by batch.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.

- **batch_size**: integer. Number of samples per gradient update.
- **verbose**: verbosity mode, 0 or 1.
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

Generates output predictions for the input samples, processing the samples in a batched way.

Arguments

- **x**: the input data, as a Numpy array.
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A Numpy array of predictions.

---

predict_classes

```
predict_classes(self, x, batch_size=32, verbose=1)
```

Generate class predictions for the input samples batch by batch.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A numpy array of class predictions.

---

predict_proba

```
predict_proba(self, x, batch_size=32, verbose=1)
```

Generates class probability predictions for the input samples batch by batch.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A Numpy array of probability predictions.

---

train_on_batch

```
train_on_batch(self, x, y, class_weight=None, sample_weight=None)
```

Single gradient update over one batch of samples.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **class_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar training loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

Evaluates the model over a single batch of samples.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict_on_batch

```
predict_on_batch(self, x)
```

Returns predictions for a single batch of samples.

---

fit_generator

```
fit_generator(self, generator, samples_per_epoch, nb_epoch, verbose=1, callbacks=[], validation
```

Fits the model on data generated batch-by-batch by a Python generator. The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

Arguments

- **generator**: a generator. The output of the generator must be either
    - a tuple (inputs, targets)
    - a tuple (inputs, targets, sample_weights). All arrays should contain the same number of samples. The generator is expected to loop over its data indefinitely. An epoch finishes when `samples_per_epoch` samples have been seen by the model.
- **samples_per_epoch**: integer, number of samples to process before going to the next epoch.
- **nb_epoch**: integer, total number of iterations on the data.
- **verbose**: verbosity mode, 0, 1, or 2.
- **callbacks**: list of callbacks to be called during training.
- **validation_data**: this can be either
    - a generator for the validation data
    - a tuple (inputs, targets)
    - a tuple (inputs, targets, sample_weights).

- **nb_val_samples**: only relevant if `validation_data` is a generator. number of samples to use from validation generator at the end of every epoch.
- **class_weight**: dictionary mapping class indices to a weight for the class.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up
- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

Returns

A `History` object.

Example

```
def generate_arrays_from_file(path):
    while 1:
    f = open(path)
    for line in f:
        # create Numpy arrays of input data
        # and labels, from each line in the file
        x, y = process_line(line)
        yield (x, y)
    f.close()

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
        samples_per_epoch=10000, nb_epoch=10)
```

evaluate_generator

```
evaluate_generator(self, generator, val_samples, max_q_size=10, nb_worker=1, pickle_safe=False)
```

Evaluates the model on a data generator. The generator should return the same kind of data as accepted by `test_on_batch`.

- **Arguments**:
- **generator**: generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights)
- **val_samples**: total number of samples to generate from `generator` before returning.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up
- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

Model class API

In the functional API, given an input tensor and output tensor, you can instantiate a `Model` via:

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(input=a, output=b)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(input=[a1, a2], output=[b1, b3, b3])
```

For a detailed introduction of what `Model` can do, read this guide to the Keras functional API.

Useful attributes of Model

- `model.layers` is a flattened list of the layers comprising the model graph.
- `model.inputs` is the list of input tensors.
- `model.outputs` is the list of output tensors.

Methods

compile

```
compile(self, optimizer, loss, metrics=[], loss_weights=None, sample_weight_mode=None)
```

Configures the model for training.

Arguments

- **optimizer**: str (name of optimizer) or optimizer object. See optimizers.
- **loss**: str (name of objective function) or objective function. See objectives. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of objectives.
- **metrics**: list of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you

could also pass a dictionary, such as `metrics={'output_a': 'accuracy'}`.

- **sample_weight_mode**: if you need to do timestep-wise sample weighting (2D weights), set this to "temporal". "None" defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **kwargs**: when using the Theano backend, these arguments are passed into K.function. Ignored for Tensorflow backend.

---

fit

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[], validation_split=0.0, vali
```

Trains the model for a fixed number of epochs (iterations on a dataset).

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **batch_size**: integer. Number of samples per gradient update.
- **nb_epoch**: integer, the number of times to iterate over the training data arrays.
- **verbose**: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = verbose, 2 = one log line per epoch.
- **callbacks**: list of callbacks to be called during training. See callbacks.
- **validation_split**: float between 0 and 1: fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch.
- **validation_data**: data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. This could be a tuple (x_val, y_val) or a tuple (val_x, val_y, val_sample_weights).
- **shuffle**: boolean, whether to shuffle the training data before each epoch.
- **class_weight**: optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

Returns

A `History` instance. Its `history` attribute contains all information collected during training.

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

Returns the loss value and metrics values for the model in test mode. Computation is done in batches.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **batch_size**: integer. Number of samples per gradient update.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict

```
predict(self, x, batch_size=32, verbose=0)
```

Generates output predictions for the input samples, processing the samples in a batched way.

Arguments

- **x**: the input data, as a Numpy array (or list of Numpy arrays if the model has multiple outputs).
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A Numpy array of predictions.

---

train_on_batch

```
train_on_batch(self, x, y, sample_weight=None, class_weight=None)
```

Runs a single gradient update on a single batch of data.

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.

Returns

Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

test_on_batch

```
test_on_batch(self, x, y, sample_weight=None)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict_on_batch

```
predict_on_batch(self, x)
```

Returns predictions for a single batch of samples.

---

fit_generator

```
fit_generator(self, generator, samples_per_epoch, nb_epoch, verbose=1, callbacks=[], validation
```

Fits the model on data generated batch-by-batch by a Python generator. The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

Arguments

- **generator**: a generator. The output of the generator must be either
    - a tuple (inputs, targets)
    - a tuple (inputs, targets, sample_weights). All arrays should contain the same number of samples. The generator is expected to loop over its data indefinitely. An epoch finishes when `samples_per_epoch` samples have been seen by the model.
- **samples_per_epoch**: integer, number of samples to process before going to the next epoch.
- **nb_epoch**: integer, total number of iterations on the data.
- **verbose**: verbosity mode, 0, 1, or 2.
- **callbacks**: list of callbacks to be called during training.
- **validation_data**: this can be either
    - a generator for the validation data
    - a tuple (inputs, targets)
    - a tuple (inputs, targets, sample_weights).
- **nb_val_samples**: only relevant if `validation_data` is a generator. number of samples to use from validation generator at the end of every epoch.
- **class_weight**: dictionary mapping class indices to a weight for the class.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up when using process based threading

- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

Returns

A `History` object.

Example

```
def generate_arrays_from_file(path):
    while 1:
    f = open(path)
    for line in f:
        # create numpy arrays of input data
        # and labels, from each line in the file
        x1, x2, y = process_line(line)
        yield ({'input_1': x1, 'input_2': x2}, {'output': y})
    f.close()

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
        samples_per_epoch=10000, nb_epoch=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, val_samples, max_q_size=10, nb_worker=1, pickle_safe=False)
```

Evaluates the model on a data generator. The generator should return the same kind of data as accepted by `test_on_batch`.

- **Arguments**:
- **generator**: generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights)
- **val_samples**: total number of samples to generate from `generator` before returning.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up when using process based threading
- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

get_layer

```
get_layer(self, name=None, index=None)
```

Returns a layer based on either its name (unique) or its index in the graph. Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: string, name of layer.
- **index**: integer, index of layer.

Returns

A layer instance.

About Keras layers

All Keras layers have a number of methods in common:

- `layer.get_weights()`: returns the weights of the layer as a list of Numpy arrays.
- `layer.set_weights(weights)`: sets the weights of the layer from a list of Numpy arrays (with the same shapes as the output of `get_weights`).
- `layer.get_config()`: returns a dictionary containing the configuration of the layer. The layer can be reinstantiated from its config via:

```
from keras.utils.layer_utils import layer_from_config

config = layer.get_config()
layer = layer_from_config(config)
```

If a layer has a single node (i.e. if it isn't a shared layer), you can get its input tensor, output tensor, input shape and output shape via:

- `layer.input`
- `layer.output`
- `layer.input_shape`
- `layer.output_shape`

If the layer has multiple nodes (see: the concept of layer node and shared layers), you can use the following methods:

- `layer.get_input_at(node_index)`
- `layer.get_output_at(node_index)`
- `layer.get_input_shape_at(node_index)`
- `layer.get_output_shape_at(node_index)`

Dense

[source]

```
keras.layers.core.Dense(output_dim, init='glorot_uniform', activation='linear', weights=None, W
```

Just your regular fully connected NN layer.

Example

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_dim=16))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# this is equivalent to the above:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

Arguments

- **output_dim**: int > 0.
- **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, output_dim)`.

Activation

[source]

```
keras.layers.core.Activation(activation)
```

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.

Dropout

[source]

```
keras.layers.core.Dropout(p)
```

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

Arguments

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

References

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

---

Flatten

[source]

```
keras.layers.core.Flatten()
```

Flattens the input. Does not affect the batch size.

Example

```
model = Sequential()
model.add(Convolution2D(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)
```

Reshape

[source]

```
keras.layers.core.Reshape(target_shape)
```

Reshapes an output to a certain shape.

Arguments

- **target_shape**: target shape. Tuple of integers, does not include the samples dimension (batch size).

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 samples 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)
```

Permute

[source]

```
keras.layers.core.Permute(dims)
```

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.

RepeatVector

[source]

```
keras.layers.core.RepeatVector(n)
```

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 `(nb_samples, features)`.

Output shape

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

Merge

[source]

```
keras.engine.topology.Merge(layers=None, mode='sum', concat_axis=-1, dot_axes=-1, output_shape=
```

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

Example usage

```
model1 = Sequential()
model1.add(Dense(32))

model2 = Sequential()
model2.add(Dense(32))

merged_model = Sequential()
merged_model.add(Merge([model1, model2], mode='concat', concat_axis=1)
- ____TODO__: would this actually work? it needs to.__

# achieve this with get_source_inputs in Sequential.
```

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`.
- **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 `get_output_shape_for` 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.

Lambda

[source]

```
keras.layers.core.Lambda(function, output_shape=None, arguments={})
```

Used for evaluating an arbitrary Theano / TensorFlow expression on the output of the previous layer.

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 one argument: the output of previous layer
- **output_shape**: Expected output shape from function. 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.

ActivityRegularization

[source]

```
keras.layers.core.ActivityRegularization(l1=0.0, l2=0.0)
```

Layer that passes through its input unchanged, but applies an update to the cost function based on the 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.

Masking

[source]

```
keras.layers.core.Masking(mask_value=0.0)
```

Masks an input sequence by using a mask value to identify timesteps to be skipped.

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 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))
```

Highway

[source]

```
keras.layers.core.Highway(init='glorot_uniform', transform_bias=-2, activation='linear', weight
```

Densely connected highway network, 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.
- **transform_bias**: value for the bias to take on initially (default -2)
- **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

MaxoutDense

[source]

```
keras.layers.core.MaxoutDense(output_dim, nb_feature=4, init='glorot_uniform', weights=None, W_
```

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

TimeDistributedDense

[source]

```
keras.layers.core.TimeDistributedDense(output_dim, init='glorot_uniform', activation='linear',
```

Apply a same Dense layer for each dimension[1] (time_dimension) input. Especially useful after a recurrent network with 'return_sequence=True'.

- **Note**: this layer is deprecated, prefer using the `TimeDistributed` wrapper:

```
model.add(TimeDistributed(Dense(32)))
```

Input shape

3D tensor with shape `(nb_sample, time_dimension, input_dim)`.

Output shape

3D tensor with shape `(nb_sample, time_dimension, output_dim)`.

Arguments

- **output_dim**: int > 0.
- **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_length**: length of inputs sequences (integer, or None for variable-length sequences).

Convolution1D

[source]

```
keras.layers.convolutional.Convolution1D(nb_filter, filter_length, init='uniform', activation='
```

Convolution operator for filtering neighborhoods of one-dimensional inputs. When using this layer as the first layer in a model, either provide the keyword argument `input_dim` (int, e.g. 128 for sequences of 128-dimensional vectors), or `input_shape` (tuple of integers, e.g. (10, 128) for sequences of 10 vectors of 128-dimensional vectors).

Example

```
# apply a convolution 1d of length 3 to a sequence with 10 timesteps,
# with 64 output filters
model = Sequential()
model.add(Convolution1D(64, 3, border_mode='same', input_shape=(10, 32)))
# now model.output_shape == (None, 10, 64)

# add a new conv1d on top
model.add(Convolution1D(32, 3, border_mode='same'))
# now model.output_shape == (None, 10, 32)
```

Arguments

- **nb_filter**: Number of convolution kernels to use (dimensionality of the output).
- **filter_length**: The extension (spatial or temporal) of each filter.
- **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.
- **border_mode**: 'valid' or 'same'.
- **subsample_length**: factor by which to subsample output.
- **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**: Number of channels/dimensions in the input. Either this argument or the keyword argument `input_shape` must be provided when using this layer as the first layer in a model.
- **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

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

Output shape

3D tensor with shape: `(samples, new_steps, nb_filter)`. `steps` value might have changed due to padding.

Convolution2D

[source]

```
keras.layers.convolutional.Convolution2D(nb_filter, nb_row, nb_col, init='glorot_uniform', acti
```

Convolution operator for filtering windows of two-dimensional inputs. 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=(3, 128, 128)` for 128x128 RGB pictures.

Examples

```
# apply a 3x3 convolution with 64 output filters on a 256x256 image:
model = Sequential()
model.add(Convolution2D(64, 3, 3, border_mode='same', input_shape=(3, 256, 256)))
# now model.output_shape == (None, 64, 256, 256)

# add a 3x3 convolution on top, with 32 output filters:
model.add(Convolution2D(32, 3, 3, border_mode='same'))
# now model.output_shape == (None, 32, 256, 256)
```

Arguments

- **nb_filter**: Number of convolution filters to use.
- **nb_row**: Number of rows in the convolution kernel.
- **nb_col**: Number of columns in the convolution kernel.
- **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.
- **border_mode**: 'valid' or 'same'.
- **subsample**: tuple of length 2. Factor by which to subsample output. Also called strides elsewhere.
- **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.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".
- **bias**: whether to include a bias (i.e. make the layer affine rather than linear).

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, rows, cols, channels)` if dim_ordering='tf'.

Output shape

4D tensor with shape: `(samples, nb_filter, new_rows, new_cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, new_rows, new_cols, nb_filter)` if dim_ordering='tf'. `rows` and `cols` values might have changed due to padding.

AtrousConvolution2D

[source]

```
keras.layers.convolutional.AtrousConvolution2D(nb_filter, nb_row, nb_col, init='glorot_uniform'
```

Atrous Convolution operator for filtering windows of two-dimensional inputs. A.k.a dilated convolution or convolution with holes. 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=(3, 128, 128)` for 128x128 RGB pictures.

Examples

```
# apply a 3x3 convolution with atrous rate 2x2 and 64 output filters on a 256x256 image:
model = Sequential()
model.add(AtrousConvolution2D(64, 3, 3, atrous_rate=(2,2), border_mode='valid', input_shape=(3,
# now the actual kernel size is dilated from 3x3 to 5x5 (3+(3-1)*(2-1)=5)
# thus model.output_shape == (None, 64, 252, 252)
```

Arguments

- **nb_filter**: Number of convolution filters to use.
- **nb_row**: Number of rows in the convolution kernel.
- **nb_col**: Number of columns in the convolution kernel.
- **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.
- **border_mode**: 'valid' or 'same'.
- **subsample**: tuple of length 2. Factor by which to subsample output. Also called strides elsewhere.
- **atrous_rate**: tuple of length 2. Factor for kernel dilation. Also called filter_dilation elsewhere.
- **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.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".
- **bias**: whether to include a bias (i.e. make the layer affine rather than linear).

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, rows, cols, channels)` if dim_ordering='tf'.

Output shape

4D tensor with shape: `(samples, nb_filter, new_rows, new_cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, new_rows, new_cols, nb_filter)` if dim_ordering='tf'. `rows` and `cols` values might have changed due to padding.

References

- Multi-Scale Context Aggregation by Dilated Convolutions

Convolution3D

[source]

```
keras.layers.convolutional.Convolution3D(nb_filter, kernel_dim1, kernel_dim2, kernel_dim3, init
```

Convolution operator for filtering windows of three-dimensional inputs. 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=(3, 10, 128, 128)` for 10 frames of 128x128 RGB pictures.

Arguments

- **nb_filter**: Number of convolution filters to use.
- **kernel_dim1**: Length of the first dimension in the convolution kernel.
- **kernel_dim2**: Length of the second dimension in the convolution kernel.
- **kernel_dim3**: Length of the third dimension in the convolution kernel.
- **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.
- **border_mode**: 'valid' or 'same'.
- **subsample**: tuple of length 3. Factor by which to subsample output. Also called strides elsewhere.
    - **Note**: 'subsample' is implemented by slicing the output of conv3d with strides=(1,1,1).
- **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.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 4. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".
- **bias**: whether to include a bias (i.e. make the layer affine rather than linear).

Input shape

5D tensor with shape: `(samples, channels, conv_dim1, conv_dim2, conv_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, conv_dim1, conv_dim2, conv_dim3, channels)` if dim_ordering='tf'.

Output shape

5D tensor with shape: `(samples, nb_filter, new_conv_dim1, new_conv_dim2, new_conv_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, new_conv_dim1, new_conv_dim2, new_conv_dim3, nb_filter)` if dim_ordering='tf'. `new_conv_dim1`, `new_conv_dim2` and `new_conv_dim3` values might have changed due to padding.

UpSampling1D

[source]

```
keras.layers.convolutional.UpSampling1D(length=2)
```

Repeat each temporal step `length` times along the time axis.

Arguments

- **length**: integer. Upsampling factor.

Input shape

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

Output shape

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

UpSampling2D

[source]

```
keras.layers.convolutional.UpSampling2D(size=(2, 2), dim_ordering='th')
```

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

Arguments

- **size**: tuple of 2 integers. The upsampling factors for rows and columns.

- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, rows, cols, channels)` if dim_ordering='tf'.

Output shape

4D tensor with shape: `(samples, channels, upsampled_rows, upsampled_cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, upsampled_rows, upsampled_cols, channels)` if dim_ordering='tf'.

UpSampling3D

[source]

```
keras.layers.convolutional.UpSampling3D(size=(2, 2, 2), dim_ordering='th')
```

Repeat the first, second and third dimension of the data by size[0], size[1] and size[2] respectively.

Arguments

- **size**: tuple of 3 integers. The upsampling factors for dim1, dim2 and dim3.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 4. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

5D tensor with shape: `(samples, channels, dim1, dim2, dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, dim1, dim2, dim3, channels)` if dim_ordering='tf'.

Output shape

5D tensor with shape: `(samples, channels, upsampled_dim1, upsampled_dim2, upsampled_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, upsampled_dim1, upsampled_dim2, upsampled_dim3, channels)` if dim_ordering='tf'.

ZeroPadding1D

[source]

```
keras.layers.convolutional.ZeroPadding1D(padding=1)
```

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

Arguments

- **padding**: int How many zeros to add at the beginning and end of the padding dimension (axis 1).

Input shape

3D tensor with shape (samples, axis_to_pad, features)

Output shape

3D tensor with shape (samples, padded_axis, features)

ZeroPadding2D

[source]

```
keras.layers.convolutional.ZeroPadding2D(padding=(1, 1), dim_ordering='th')
```

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

Arguments

- **padding**: tuple of int (length 2) How many zeros to add at the beginning and end of the 2 padding dimensions (axis 3 and 4).
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

4D tensor with shape: (samples, depth, first_axis_to_pad, second_axis_to_pad)

Output shape

4D tensor with shape: (samples, depth, first_padded_axis, second_padded_axis)

ZeroPadding3D

[source]

```
keras.layers.convolutional.ZeroPadding3D(padding=(1, 1, 1), dim_ordering='th')
```

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

Arguments

- **padding**: tuple of int (length 3) How many zeros to add at the beginning and end of the 3 padding dimensions (axis 3, 4 and 5).
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 4. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

5D tensor with shape: (samples, depth, first_axis_to_pad, second_axis_to_pad, third_axis_to_pad)

Output shape

5D tensor with shape: (samples, depth, first_padded_axis, second_padded_axis, third_axis_to_pad)

MaxPooling1D

[source]

```
keras.layers.pooling.MaxPooling1D(pool_length=2, stride=None, border_mode='valid')
```

Max pooling operation for temporal data.

Input shape

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

Output shape

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

Arguments

- **pool_length**: factor by which to downscale. 2 will halve the input.
- **stride**: integer, or None. Stride value. If None, it will default to `pool_length`.
- **border_mode**: 'valid' or 'same'.
  - **Note**: 'same' will only work with TensorFlow for the time being.

MaxPooling2D

[source]

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None, border_mode='valid', dim_orde
```

Max pooling operation for spatial data.

Arguments

- **pool_size**: tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.
- **strides**: tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **border_mode**: 'valid' or 'same'.
  - **Note**: 'same' will only work with TensorFlow for the time being.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, rows, cols, channels)` if dim_ordering='tf'.

Output shape

4D tensor with shape: `(nb_samples, channels, pooled_rows, pooled_cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, pooled_rows, pooled_cols, channels)` if dim_ordering='tf'.

MaxPooling3D

[source]

```
keras.layers.pooling.MaxPooling3D(pool_size=(2, 2, 2), strides=None, border_mode='valid', dim_o
```

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.
- **border_mode**: 'valid' or 'same'.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 4. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

5D tensor with shape: `(samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, len_pool_dim1, len_pool_dim2, len_pool_dim3, channels)` if dim_ordering='tf'.

Output shape

5D tensor with shape: `(nb_samples, channels, pooled_dim1, pooled_dim2, pooled_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, pooled_dim1, pooled_dim2, pooled_dim3, channels)` if dim_ordering='tf'.

AveragePooling1D

[source]

```
keras.layers.pooling.AveragePooling1D(pool_length=2, stride=None, border_mode='valid')
```

Average pooling for temporal data.

Arguments

- **pool_length**: factor by which to downscale. 2 will halve the input.
- **stride**: integer, or None. Stride value. If None, it will default to `pool_length`.
- **border_mode**: 'valid' or 'same'.
  - **Note**: 'same' will only work with TensorFlow for the time being.

Input shape

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

Output shape

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

AveragePooling2D

[source]

```
keras.layers.pooling.AveragePooling2D(pool_size=(2, 2), strides=None, border_mode='valid', dim_
```

Average pooling operation for spatial data.

Arguments

- **pool_size**: tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.
- **strides**: tuple of 2 integers, or None. Strides values. If None, it will default to `pool_size`.
- **border_mode**: 'valid' or 'same'.
  - **Note**: 'same' will only work with TensorFlow for the time being.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 3. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

4D tensor with shape: `(samples, channels, rows, cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, rows, cols, channels)` if dim_ordering='tf'.

Output shape

4D tensor with shape: `(nb_samples, channels, pooled_rows, pooled_cols)` if dim_ordering='th' or 4D tensor with shape: `(samples, pooled_rows, pooled_cols, channels)` if dim_ordering='tf'.

AveragePooling3D

[source]

```
keras.layers.pooling.AveragePooling3D(pool_size=(2, 2, 2), strides=None, border_mode='valid', d
```

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.
- **border_mode**: 'valid' or 'same'.
- **dim_ordering**: 'th' or 'tf'. In 'th' mode, the channels dimension (the depth) is at index 1, in 'tf' mode is it at index 4. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

Input shape

5D tensor with shape: `(samples, channels, len_pool_dim1, len_pool_dim2, len_pool_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, len_pool_dim1, len_pool_dim2, len_pool_dim3, channels)` if dim_ordering='tf'.

Output shape

5D tensor with shape: `(nb_samples, channels, pooled_dim1, pooled_dim2, pooled_dim3)` if dim_ordering='th' or 5D tensor with shape: `(samples, pooled_dim1, pooled_dim2, pooled_dim3, channels)` if dim_ordering='tf'.

Recurrent

[source]

```
keras.layers.recurrent.Recurrent(weights=None, return_sequences=False, go_backwards=False, stat
```

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.

# the following is identical:
model = Sequential()
model.add(LSTM(32, input_dim=64, input_length=10))

# for subsequent layers, not need to specify the input size:
model.add(LSTM(16))
```

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.
- **go_backwards**: Boolean (default False). If True, process 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.
- **unroll**: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. When using TensorFlow, the network is always unrolled, so this argument does not do anything. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **consume_less**: one of "cpu", "mem", or "gpu" (LSTM/GRU only). If set to "cpu", the RNN will use an implementation that uses fewer, larger matrix products, thus running faster on CPU but consuming more memory. If set to "mem", 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 "gpu" (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 shape

3D tensor with shape `(nb_samples, timesteps, input_dim)`.

Output shape

- if `return_sequences`: 3D tensor with shape `(nb_samples, timesteps, output_dim)`.
- else, 2D tensor with shape `(nb_samples, output_dim)`.

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`.

TensorFlow warning

For the time being, when using the TensorFlow backend, the number of timesteps used must be specified in your model. Make sure to pass an `input_length` int argument to your recurrent layer (if it comes first in your model), or to pass a complete `input_shape` argument to the first layer in your model otherwise.

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_shape=(...)` 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.

Note on using dropout with TensorFlow

When using the TensorFlow backend, specify a fixed batch size for your model following the notes on statefulness RNNs.

SimpleRNN

[source]

```
keras.layers.recurrent.SimpleRNN(output_dim, init='glorot_uniform', inner_init='orthogonal', ac
```

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

Arguments

- **output_dim**: dimension of the internal projections and the final output.
- **init**: weight initialization function. Can be the name of an existing function (str), or a Theano function (see: initializations).
- **inner_init**: initialization function of the inner cells.
- **activation**: activation function. Can be the name of an existing function (str), or a Theano function (see: activations).
- **W_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the input weights matrices.
- **U_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the recurrent weights matrices.
- **b_regularizer**: instance of WeightRegularizer, applied to the bias.
- **dropout_W**: float between 0 and 1. Fraction of the input units to drop for input gates.
- **dropout_U**: float between 0 and 1. Fraction of the input units to drop for recurrent connections.

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

GRU

[source]

```
keras.layers.recurrent.GRU(output_dim, init='glorot_uniform', inner_init='orthogonal', activati
```

Gated Recurrent Unit - Cho et al. 2014.

Arguments

- **output_dim**: dimension of the internal projections and the final output.
- **init**: weight initialization function. Can be the name of an existing function (str), or a Theano function (see: initializations).
- **inner_init**: initialization function of the inner cells.
- **activation**: activation function. Can be the name of an existing function (str), or a Theano function (see: activations).
- **inner_activation**: activation function for the inner cells.
- **W_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the input weights matrices.
- **U_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the recurrent weights matrices.
- **b_regularizer**: instance of WeightRegularizer, applied to the bias.
- **dropout_W**: float between 0 and 1. Fraction of the input units to drop for input gates.
- **dropout_U**: float between 0 and 1. Fraction of the input units to drop for recurrent connections.

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

---

LSTM

[source]

```
keras.layers.recurrent.LSTM(output_dim, init='glorot_uniform', inner_init='orthogonal', forget_
```

Long-Short Term Memory unit - Hochreiter 1997.

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

Arguments

- **output_dim**: dimension of the internal projections and the final output.
- **init**: weight initialization function. Can be the name of an existing function (str), or a Theano function (see: initializations).
- **inner_init**: initialization function of the inner cells.
- **forget_bias_init**: initialization function for the bias of the forget gate. Jozefowicz et al. recommend initializing with ones.
- **activation**: activation function. Can be the name of an existing function (str), or a Theano function (see: activations).

- **inner_activation**: activation function for the inner cells.
- **W_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the input weights matrices.
- **U_regularizer**: instance of WeightRegularizer (eg. L1 or L2 regularization), applied to the recurrent weights matrices.
- **b_regularizer**: instance of WeightRegularizer, applied to the bias.
- **dropout_W**: float between 0 and 1. Fraction of the input units to drop for input gates.
- **dropout_U**: float between 0 and 1. Fraction of the input units to drop for recurrent connections.

References

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

Embedding

[source]

```
keras.layers.embeddings.Embedding(input_dim, output_dim, init='uniform', input_length=None, W_r
```

Turn 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
  # 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, ie. 1 + maximum integer index occurring in the input data.
- **output_dim**: int >= 0. Dimension of the dense embedding.
- **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 1 element, of shape `(input_dim, output_dim)`.
- **W_regularizer**: instance of the regularizers module (eg. L1 or L2 regularization), applied to the embedding matrix.
- **W_constraint**: instance of the constraints module (eg. maxnorm, nonneg), applied to the embedding matrix.
- **mask_zero**: Whether or not the input value 0 is a special "padding" value that should be masked out. This is useful for 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 |vocabulary| + 2).
- **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).
- **dropout**: float between 0 and 1. Fraction of the embeddings to drop.

Input shape

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

Output shape

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

References

- A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

LeakyReLU

[source]

```
keras.layers.advanced_activations.LeakyReLU(alpha=0.3)
```

Special version of a Rectified Linear Unit that 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.

PReLU

[source]

```
keras.layers.advanced_activations.PReLU(init='zero', weights=None)
```

Parametric Rectified Linear Unit: `f(x) = alphas * x for x < 0`, `f(x) = x for x >= 0`, where `alphas` 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

- **init**: initialization function for the weights.
- **weights**: initial weights, as a list of a single Numpy array.

References

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

---

ELU

[source]

```
keras.layers.advanced_activations.ELU(alpha=1.0)
```

Exponential Linear Unit: `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)

ParametricSoftplus

[source]

```
keras.layers.advanced_activations.ParametricSoftplus(alpha_init=0.2, beta_init=5.0, weights=Non
```

Parametric Softplus: `alpha * log(1 + exp(beta * 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_init**: float. Initial value of the alpha weights.
- **beta_init**: float. Initial values of the beta weights.
- **weights**: initial weights, as a list of 2 numpy arrays.

References

- Inferring Nonlinear Neuronal Computation Based on Physiologically Plausible Inputs

---

ThresholdedReLU

[source]

```
keras.layers.advanced_activations.ThresholdedReLU(theta=1.0)
```

Thresholded Rectified Linear Unit: `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

---

SReLU

[source]

```
keras.layers.advanced_activations.SReLU(t_left_init='zero', a_left_init='glorot_uniform', t_rig
```

S-shaped Rectified Linear Unit.

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

- **t_left_init**: initialization function for the left part intercept
- **a_left_init**: initialization function for the left part slope
- **t_right_init**: initialization function for the right part intercept
- **a_right_init**: initialization function for the right part slope

References

- Deep Learning with S-shaped Rectified Linear Activation Units

BatchNormalization

[source]

```
keras.layers.normalization.BatchNormalization(epsilon=1e-06, mode=0, axis=-1, momentum=0.99, we
```

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

- **epsilon**: small float > 0. Fuzz parameter.
- **mode**: integer, 0, 1 or 2.
  - 0: feature-wise normalization. Each feature map in the input will be normalized separately. The axis on which to normalize is specified by the `axis` argument. Note that if the input is a 4D image tensor using Theano conventions (samples, channels, rows, cols) then you should set `axis` to `1` to normalize along the channels axis. During training we use per-batch statistics to normalize the data, and during testing we use running averages computed during the training phase.
  - 1: sample-wise normalization. This mode assumes a 2D input.
  - 2: feature-wise normalization, like mode 0, but using per-batch statistics to normalize the data during both testing and training.
- **axis**: integer, axis along which to normalize in mode 0. For instance, if your input tensor has shape (samples, channels, rows, cols), set axis to 1 to normalize per feature map (channels axis).
- **momentum**: momentum in the computation of the exponential average of the mean and standard deviation of the data, for feature-wise normalization.
- **weights**: Initialization weights. List of 2 Numpy arrays, with shapes: `[(input_shape,), (input_shape,)]` Note that the order of this list is [gamma, beta, mean, std]
- **beta_init**: name of initialization function for shift parameter (see initializations), or alternatively, Theano/TensorFlow function to use for weights initialization. This parameter is only relevant if you don't pass a `weights` argument.
- **gamma_init**: name of initialization function for scale parameter (see initializations), or alternatively, Theano/TensorFlow function to use for weights initialization. This parameter is only relevant if you don't pass a `weights` argument.

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

GaussianNoise

[source]

```
keras.layers.noise.GaussianNoise(sigma)
```

Apply to the input an additive zero-centered Gaussian noise with standard deviation `sigma`. This is useful to mitigate overfitting (you could see it as a kind 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

- **sigma**: 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.

GaussianDropout

[source]

```
keras.layers.noise.GaussianDropout(p)
```

Apply to the input an multiplicative one-centered Gaussian noise with standard deviation `sqrt(p/(1-p))`.

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

Arguments

- **p**: float, drop probability (as with `Dropout`).

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

TimeDistributed

[source]

```
keras.layers.wrappers.TimeDistributed(layer)
```

This wrapper allows to apply 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)

# subsequent layers: no need for input_shape
model.add(TimeDistributed(Dense(32)))
# now model.output_shape == (None, 10, 32)
```

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

Note this is strictly equivalent to using `layers.core.TimeDistributedDense`. However what is different about `TimeDistributed` is that it can be used with arbitrary layers, not just `Dense`, for instance with a `Convolution2D` layer:

```
model = Sequential()
model.add(TimeDistributed(Convolution2D(64, 3, 3), input_shape=(10, 3, 299, 299)))
```

Arguments

- **layer**: a layer instance.

Writing your own Keras layers

For simple, stateless custom operations, you are probably better off using `layers.core.Lambda` layers. But for any custom operation that has trainable weights, you should implement your own layer.

Here is the skeleton of a Keras layer. There are only three methods you need to implement:

- `build(input_shape)`: this is where you will define your weights. Trainable weights should be added to the list `self.trainable_weights`. Other attributes of note are: `self.non_trainable_weights` (list) and `self.updates` (list of update tuples (tensor, new_tensor)). For an example of how to use `non_trainable_weights` and `updates`, see the code for the `BatchNormalization` layer.
- `call(x)`: this is where the layer's logic lives. Unless you want your layer to support masking, you only have to care about the first argument passed to `call`: the input tensor.
- `get_output_shape_for(input_shape)`: in case your layer modifies the shape of its input, you should specify here the shape transformation logic. This allows Keras to do automatic shape inference.

```python
from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

class MyLayer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        input_dim = input_shape[1]
        initial_weight_value = np.random.random((input_dim, output_dim))
        self.W = K.variable(initial_weight_value)
        self.trainable_weights = [self.W]

    def call(self, x, mask=None):
        return K.dot(x, self.W)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0], self.output_dim)
```

The existing Keras layers provide ample examples of how to implement almost anything. Never hesitate to read the source code!

pad_sequences

```
keras.preprocessing.sequence.pad_sequences(sequences, maxlen=None, dtype='int32')
```

Transform a list of `nb_samples sequences` (lists of scalars) into a 2D Numpy array of shape `(nb_samples, nb_timesteps)`. `nb_timesteps` is either the `maxlen` argument if provided, or the length of the longest sequence otherwise. Sequences that are shorter than `nb_timesteps` are padded with zeros at the end.

- **Return**: 2D Numpy array of shape `(nb_samples, nb_timesteps)`.
- **Arguments**:
    - **sequences**: List of lists of int or float.
    - **maxlen**: None or int. Maximum sequence length, longer sequences are truncated and shorter sequences are padded with zeros at the end.
    - **dtype**: datatype of the Numpy array returned.
    - **padding**: 'pre' or 'post', pad either before or after each sequence.
    - **truncating**: 'pre' or 'post', remove values from sequences larger than maxlen either in the beginning or in the end of the sequence
    - **value**: float, value to pad the sequences to the desired value.

skipgrams

```
keras.preprocessing.sequence.skipgrams(sequence, vocabulary_size,
    window_size=4, negative_samples=1., shuffle=True,
    categorical=False, sampling_table=None)
```

Transforms a sequence of word indexes (list of int) into couples of the form:

- (word, word in the same window), with label 1 (positive samples).
- (word, random word from the vocabulary), with label 0 (negative samples).

Read more about Skipgram in this gnomic paper by Mikolov et al.: Efficient Estimation of Word Representations in Vector Space

- **Return**: tuple `(couples, labels)`.
    - `couples` is a list of 2-elements lists of int: `[word_index, other_word_index]`.
    - `labels` is a list of 0 and 1, where 1 indicates that `other_word_index` was found in the same window

    as `word_index`, and 0 indicates that `other_word_index` was random.
    - if categorical is set to True, the labels are categorical, ie. 1 becomes [0,1], and 0 becomes [1, 0].
- **Arguments**:
    - **sequence**: list of int indexes. If using a sampling_table, the index of a word should be its the rank in the dataset (starting at 1).
    - **vocabulary_size**: int.
    - **window_size**: int. maximum distance between two words in a positive couple.
    - **negative_samples**: float >= 0. 0 for no negative (=random) samples. 1 for same number as positive samples. etc.
    - **shuffle**: boolean. Whether to shuffle the samples.
    - **categorical**: boolean. Whether to make the returned labels categorical.
    - **sampling_table**: Numpy array of shape `(vocabulary_size,)` where `sampling_table[i]` is the probability of sampling the word with index i (assumed to be i-th most common word in the dataset).

---

make_sampling_table

```
keras.preprocessing.sequence.make_sampling_table(size, sampling_factor=1e-5)
```

Used for generating the `sampling_table` argument for `skipgrams`. `sampling_table[i]` is the probability of sampling the word i-th most common word in a dataset (more common words should be sampled less frequently, for balance).

- **Return**: Numpy array of shape `(size,)`.
- **Arguments**:
    - **size**: size of the vocabulary considered.
    - **sampling_factor**: lower values result in a longer probability decay (common words will be sampled less frequently). If set to 1, no subsampling will be performed (all sampling probabilities will be 1).

text_to_word_sequence

```
keras.preprocessing.text.text_to_word_sequence(text,
    filters=base_filter(), lower=True, split=" ")
```

Split a sentence into a list of words.

- **Return**: List of words (str).
- **Arguments**:
    - **text**: str.
    - **filters**: list (or concatenation) of characters to filter out, such as punctuation. Default: base_filter(), includes basic punctuation, tabs, and newlines.
    - **lower**: boolean. Whether to set the text to lowercase.
    - **split**: str. Separator for word splitting.

one_hot

```
keras.preprocessing.text.one_hot(text, n,
    filters=base_filter(), lower=True, split=" ")
```

One-hot encode a text into a list of word indexes in a vocabulary of size n.

- **Return**: List of integers in [1, n]. Each integer encodes a word (unicity non-guaranteed).
- **Arguments**: Same as `text_to_word_sequence` above.
    - **n**: int. Size of vocabulary.

Tokenizer

```
keras.preprocessing.text.Tokenizer(nb_words=None, filters=base_filter(),
    lower=True, split=" ")
```

Class for vectorizing texts, or/and turning texts into sequences (=list of word indexes, where the word of rank i in the dataset (starting at 1) has index i).

- **Arguments**: Same as `text_to_word_sequence` above.
    - **nb_words**: None or int. Maximum number of words to work with (if set, tokenization will be restricted to the top nb_words most common words in the dataset).
- **Methods**:

    - **fit_on_texts(texts)**:
        - **Arguments**:
            - **texts**: list of texts to train on.
    - **texts_to_sequences(texts)**
        - **Arguments**:
            - **texts**: list of texts to turn to sequences.
        - **Return**: list of sequences (one per text input).
    - **texts_to_sequences_generator(texts)**: generator version of the above.
        - **Return**: yield one sequence per input text.
    - **texts_to_matrix(texts)**:
        - **Return**: numpy array of shape `(len(texts), nb_words)`.
        - **Arguments**:
            - **texts**: list of texts to vectorize.
            - **mode**: one of "binary", "count", "tfidf", "freq" (default: "binary").
    - **fit_on_sequences(sequences)**:
        - **Arguments**:
            - **sequences**: list of sequences to train on.
    - **sequences_to_matrix(sequences)**:
        - **Return**: numpy array of shape `(len(sequences), nb_words)`.
        - **Arguments**:
            - **sequences**: list of sequences to vectorize.
            - **mode**: one of "binary", "count", "tfidf", "freq" (default: "binary").
- **Attributes**:
    - **word_counts**: dictionary mapping words (str) to the number of times they appeared on during fit. Only set after fit_on_texts was called.
    - **word_docs**: dictionary mapping words (str) to the number of documents/texts they appeared on during fit. Only set after fit_on_texts was called.
    - **word_index**: dictionary mapping words (str) to their rank/index (int). Only set after fit_on_texts was called.
    - **document_count**: int. Number of documents (texts/sequences) the tokenizer was trained on. Only set after fit_on_texts or fit_on_sequences was called.

ImageDataGenerator

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=0.,
    width_shift_range=0.,
    height_shift_range=0.,
    shear_range=0.,
    zoom_range=0.,
    channel_shift_range=0.,
    fill_mode='nearest',
    cval=0.,
    horizontal_flip=False,
    vertical_flip=False,
    rescale=None,
    dim_ordering=K.image_dim_ordering())
```

Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches) indefinitely.

- **Arguments**:
  - **featurewise_center**: Boolean. Set input mean to 0 over the dataset.
  - **samplewise_center**: Boolean. Set each sample mean to 0.
  - **featurewise_std_normalization**: Boolean. Divide inputs by std of the dataset.
  - **samplewise_std_normalization**: Boolean. Divide each input by its std.
  - **zca_whitening**: Boolean. Apply ZCA whitening.
  - **rotation_range**: Int. Degree range for random rotations.
  - **width_shift_range**: Float (fraction of total width). Range for random horizontal shifts.
  - **height_shift_range**: Float (fraction of total height). Range for random vertical shifts.
  - **shear_range**: Float. Shear Intensity (Shear angle in counter-clockwise direction as radians)
  - **zoom_range**: Float or [lower, upper]. Range for random zoom. If a float, `[lower, upper] = [1-zoom_range, 1+zoom_range]`.
  - **channel_shift_range**: Float. Range for random channel shifts.
  - **fill_mode**: One of {"constant", "nearest", "reflect" or "wrap"}. Points outside the boundaries of the input are filled according to the given mode.
  - **cval**: Float or Int. Value used for points outside the boundaries when `fill_mode = "constant"`.
  - **horizontal_flip**: Boolean. Randomly flip inputs horizontally.
  - **vertical_flip**: Boolean. Randomly flip inputs vertically.
  - **rescale**: rescaling factor. Defaults to None. If None or 0, no rescaling is applied, otherwise we multiply the data by the value provided (before applying any other transformation).

- **dim_ordering**: One of {"th", "tf"}. "tf" mode means that the images should have shape `(samples, width, height, channels)`, "th" mode means that the images should have shape `(samples, channels, width, height)`. It defaults to the `image_dim_ordering` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "th".

- **Methods**:
  - **fit(X)**: Compute the internal data stats related to the data-dependent transformations, based on an array of sample data. Only required if featurewise_center or featurewise_std_normalization or zca_whitening.
    - **Arguments**:
      - **X**: sample data.
      - **augment**: Boolean (default: False). Whether to fit on randomly augmented samples.
      - **rounds**: int (default: 1). If augment, how many augmentation passes over the data to use.
  - **flow(X, y)**: Takes numpy data & label arrays, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.
    - **Arguments**:
      - **X**: data.
      - **y**: labels.
      - **batch_size**: int (default: 32).
      - **shuffle**: boolean (defaut: False).
      - **save_to_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).
      - **save_prefix**: str (default: `''`). Prefix to use for filenames of saved pictures (only relevant if `save_to_dir` is set).
      - **save_format**: one of "png", "jpeg" (only relevant if `save_to_dir` is set). Default: "jpeg".
    - **_yields**: Tuples of `(x, y)` where `x` is a numpy array of image data and `y` is a numpy array of corresponding labels. The generator loops indefinitely.
  - **flow_from_directory(directory)**: Takes the path to a directory, and generates batches of augmented/normalized data. Yields batches indefinitely, in an infinite loop.
    - **Arguments**:
      - __directory: path to the target directory. It should contain one subdirectory per class, and the subdirectories should contain PNG or JPG images. See this script for more details.
      - **target_size**: tuple of integers, default: `(256, 256)`. The dimensions to which all images found will be resized.
      - **color_mode**: one of "grayscale", "rbg". Default: "rgb". Whether the images will be converted to have 1 or 3 color channels.
      - **classes**: optional list of class subdirectories (e.g. `['dogs', 'cats']`). Default: None. If not provided, the list of classes will be automatically inferred (and the order of the classes, which will map to the label indices, will be alphanumeric).
      - **class_mode**: one of "categorical", "binary", "sparse" or None. Default: "categorical". Determines the type of label arrays that are returned: "categorical" will be 2D one-hot encoded labels,

"binary" will be 1D binary labels, "sparse" will be 1D integer labels. If None, no labels are returned (the generator will only yield batches of image data, which is useful to use `model.predict_generator()`, `model.evaluate_generator()`, etc.).
    - **batch_size**: size of the batches of data (default: 32).
    - **shuffle**: whether to shuffle the data (default: True)
    - **seed**: optional random seed for shuffling.
    - **save_to_dir**: None or str (default: None). This allows you to optimally specify a directory to which to save the augmented pictures being generated (useful for visualizing what you are doing).
    - **save_prefix**: str. Prefix to use for filenames of saved pictures (only relevant if `save_to_dir` is set).
    - **save_format**: one of "png", "jpeg" (only relevant if `save_to_dir` is set). Default: "jpeg".
- **Examples**:

Example of using `.flow(X, y)`:

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data(test_split=0.1)
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(X_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(X_train, Y_train, batch_size=32),
                    samples_per_epoch=len(X_train), nb_epoch=nb_epoch)

# here's a more "manual" example
for e in range(nb_epoch):
    print 'Epoch', e
    batches = 0
    for X_batch, Y_batch in datagen.flow(X_train, Y_train, batch_size=32):
        loss = model.train(X_batch, Y_batch)
        batches += 1
        if batches >= len(X_train) / 32:
            # we need to break the loop by hand because
            # the generator loops indefinitely
            break
```

Example of using `.flow_from_directory(directory)`:

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

model.fit_generator(
        train_generator,
        samples_per_epoch=2000,
        nb_epoch=50,
        validation_data=validation_generator,
        nb_val_samples=800)
```

Usage of objectives

An objective function (or loss function, or optimization score function) is one of the two parameters required to compile a model:

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

You can either pass the name of an existing objective, or pass a Theano/TensorFlow symbolic function that returns a scalar for each data-point and takes the following two arguments:

- **y_true**: True labels. Theano/TensorFlow tensor.
- **y_pred**: Predictions. Theano/TensorFlow tensor of the same shape as y_true.

The actual optimized objective is the mean of the output array across all datapoints.

For a few examples of such functions, check out the objectives source.

Available objectives

- **mean_squared_error / mse**
- **mean_absolute_error / mae**
- **mean_absolute_percentage_error / mape**
- **mean_squared_logarithmic_error / msle**
- **squared_hinge**
- **hinge**
- **binary_crossentropy**: Also known as logloss.
- **categorical_crossentropy**: Also known as multiclass logloss. **Note**: using this objective requires that your labels are binary arrays of shape `(nb_samples, nb_classes)`.
- **sparse_categorical_crossentropy**: As above but accepts sparse labels. **Note**: this objective still requires that your labels have the same number of dimensions as your outputs; you may need to add a length-1 dimension to the shape of your labels, e.g with `np.expand_dims(y, -1)`.
- **kullback_leibler_divergence / kld**: Information gain from a predicted probability distribution Q to a true probability distribution P. Gives a measure of difference between both distributions.
- **poisson**: Mean of `(predictions - targets * log(predictions))`
- **cosine_proximity**: The opposite (negative) of the mean cosine proximity between predictions and targets.

Usage of optimizers

An optimizer is one of the two arguments required for compiling a Keras model:

```
model = Sequential()
model.add(Dense(64, init='uniform', input_dim=10))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

You can either instantiate an optimizer before passing it to `model.compile()` , as in the above example, or you can call it by its name. In the latter case, the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='mean_squared_error', optimizer='sgd')
```

SGD

[source]

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

Stochastic gradient descent, with support for momentum, learning rate decay, and Nesterov momentum.

Arguments

- **lr**: float >= 0. Learning rate.
- **momentum**: float >= 0. Parameter updates momentum.
- **decay**: float >= 0. Learning rate decay over each update.
- **nesterov**: boolean. Whether to apply Nesterov momentum.

RMSprop

[source]

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08)
```

RMSProp optimizer.

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

Arguments

- **lr**: float >= 0. Learning rate.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor.

Adagrad

[source]

```
keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08)
```

Adagrad optimizer.

It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate.
- **epsilon**: float >= 0.

Adadelta

[source]

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=1e-08)
```

Adadelta optimizer.

It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate. It is recommended to leave it at the default value.
- **rho**: float >= 0.
- **epsilon**: float >= 0. Fuzz factor.

References

- Adadelta - an adaptive learning rate method

Adam

[source]

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Adam optimizer.

Default parameters follow those provided in the original paper.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor.

References

- Adam - A Method for Stochastic Optimization

Adamax

[source]

```
keras.optimizers.Adamax(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08)
```

Adamax optimizer from Adam paper's Section 7. It is a variant of Adam based on the infinity norm.

Default parameters follow those provided in the paper.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor.

References

- Adam - A Method for Stochastic Optimization

Nadam

[source]

```
keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08, schedule_decay=0.004)
```

Nesterov Adam optimizer: Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.

Default parameters follow those provided in the paper. It is recommended to leave the parameters of this optimizer at their default values.

Arguments

- **lr**: float >= 0. Learning rate.
- **beta_1/beta_2**: floats, 0 < beta < 1. Generally close to 1.
- **epsilon**: float >= 0. Fuzz factor.

References

[1] Nadam report - http://cs229.stanford.edu/proj2015/054_report.pdf [2] On the importance of initialization and momentum in deep learning - - **http**://www.cs.toronto.edu/~fritz/absps/momentum.pdf

Usage of activations

Activations can either be used through an `Activation` layer, or through the `activation` argument supported by all forward layers:

```
from keras.layers.core import Activation, Dense

model.add(Dense(64))
model.add(Activation('tanh'))
```

is equivalent to:

```
model.add(Dense(64, activation='tanh'))
```

You can also pass an element-wise Theano/TensorFlow function as an activation:

```
from keras import backend as K

def tanh(x):
    return K.tanh(x)

model.add(Dense(64, activation=tanh))
model.add(Activation(tanh))
```

Available activations

- **softmax**: Softmax applied across inputs last dimension. Expects shape either `(nb_samples, nb_timesteps, nb_dims)` or `(nb_samples, nb_dims)`.
- **softplus**
- **softsign**
- **relu**
- **tanh**
- **sigmoid**
- **hard_sigmoid**
- **linear**

On Advanced Activations

Activations that are more complex than a simple Theano/TensorFlow function (eg. learnable activations, configurable activations, etc.) are available as Advanced Activation layers, and can be found in the module `keras.layers.advanced_activations`. These include PReLU and LeakyReLU.

Usage of callbacks

A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training. You can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of the `Sequential` model. The relevant methods of the callbacks will then be called at each stage of the training.

BaseLogger

[source]

```
keras.callbacks.BaseLogger()
```

Callback that accumulates epoch averages of the metrics being monitored.

This callback is automatically applied to every Keras model.

Callback

[source]

```
keras.callbacks.Callback()
```

Abstract base class used to build new callbacks.

Properties

- **params**: dict. Training parameters (eg. verbosity, batch size, number of epochs...).
- **model**: instance of `keras.models.Model`. Reference of the model being trained.

The `logs` dictionary that callback methods take as argument will contain keys for quantities relevant to the current batch or epoch.

Currently, the `.fit()` method of the `Sequential` model class will include the following quantities in the `logs` that it passes to its callbacks:

- **on_epoch_end**: logs include `acc` and `loss`, and optionally include `val_loss` (if validation is enabled in `fit`), and `val_acc` (if validation and accuracy monitoring are enabled).
- **on_batch_begin**: logs include `size`, the number of samples in the current batch.
- **on_batch_end**: logs include `loss`, and optionally `acc` (if accuracy monitoring is enabled).

ProgbarLogger

[source]

```
keras.callbacks.ProgbarLogger()
```

Callback that prints metrics to stdout.

History

[source]

```
keras.callbacks.History()
```

Callback that records events into a `History` object.

This callback is automatically applied to every Keras model. The `History` object gets returned by the `fit` method of models.

ModelCheckpoint

[source]

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False,
```

Save the model after every epoch.

`filepath` can contain named formatting options, which will be filled the value of `epoch` and keys in `logs` (passed in `on_epoch_end`).

For example: if `filepath` is `weights.{epoch:02d}-{val_loss:.2f}.hdf5`, then multiple files will be save with the epoch number and the validation loss.

Arguments

- **filepath**: string, path to save the model file.
- **monitor**: quantity to monitor.
- **verbose**: verbosity mode, 0 or 1.
- **save_best_only**: if `save_best_only=True`, the latest best model according to the validation loss will not be overwritten.
- **mode**: one of {auto, min, max}. If `save_best_only=True`, the decision to overwrite the current save file is made based on either the maximization or the minization of the monitored. For `val_acc`, this should be `max`, for `val_loss` this should be `min`, etc. In `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **save_weights_only**: if True, then only the model's weights will be saved

(`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).

EarlyStopping

[source]

```
keras.callbacks.EarlyStopping(monitor='val_loss', patience=0, verbose=0, mode='auto')
```

Stop training when a monitored quantity has stopped improving.

Arguments

- **monitor**: quantity to be monitored.
- **patience**: number of epochs with no improvement after which training will be stopped.
- **verbose**: verbosity mode.
- **mode**: one of {auto, min, max}. In 'min' mode, training will stop when the quantity monitored has stopped decreasing; in 'max' mode it will stop when the quantity monitored has stopped increasing.

RemoteMonitor

[source]

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000', path='/publish/epoch/end/', field='
```

Callback used to stream events to a server.

Requires the `requests` library.

Arguments

- **root**: root url to which the events will be sent (at the end of every epoch). Events are sent to `root + '/publish/epoch/end/'` by default. Calls are HTTP POST, with a `data` argument which is a JSON-encoded dictionary of event data.

LearningRateScheduler

[source]

```
keras.callbacks.LearningRateScheduler(schedule)
```

Learning rate scheduler.

Arguments

- **schedule**: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

TensorBoard

[source]

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, write_graph=True)
```

Tensorboard basic visualizations.

This callback writes a log for TensorBoard, which allows you to visualize dynamic graphs of your training and test metrics, as well as activation histograms for the different layers in your model.

TensorBoard is a visualization tool provided with TensorFlow.

If you have installed TensorFlow with pip, you should be able to launch TensorBoard from the command line:

```
tensorboard --logdir=/full_path_to_your_logs
```

You can find more information about TensorBoard - __here.

Arguments

- **log_dir**: the path of the directory where to save the log files to be parsed by Tensorboard
- **histogram_freq**: frequency (in epochs) at which to compute activation histograms for the layers of the model. If set to 0, histograms won't be computed.
- **write_graph**: whether to visualize the graph in Tensorboard. The log file can become quite large when write_graph is set to True.

Create a callback

You can create a custom callback by extending the base class `keras.callbacks.Callback`. A callback has access to its associated model through the class property `self.model`.

Here's a simple example saving a list of losses over each batch during training:

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
```

Example: recording loss history

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0, callbacks=[history])

print history.losses
# outputs
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617, 0.25901699725311789]
'''
```

Example: model checkpoints

```
from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

'''
saves the model weights after each epoch if the validation loss decreased
'''
checkpointer = ModelCheckpoint(filepath="/tmp/weights.hdf5", verbose=1, save_best_only=True)
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0, validation_data=(X_test, Y_
```

Datasets

CIFAR10 small image classification

Dataset of 50,000 32x32 color training images, labeled over 10 categories, and 10,000 test images.

Usage:

```
from keras.datasets import cifar10

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

- **Return:**
  - 2 tuples:
    - **X_train, X_test**: uint8 array of RGB image data with shape (nb_samples, 3, 32, 32).
    - **y_train, y_test**: uint8 array of category labels (integers in range 0-9) with shape (nb_samples,).

CIFAR100 small image classification

Dataset of 50,000 32x32 color training images, labeled over 100 categories, and 10,000 test images.

Usage:

```
from keras.datasets import cifar100

(X_train, y_train), (X_test, y_test) = cifar100.load_data(label_mode='fine')
```

- **Return:**
  - 2 tuples:
    - **X_train, X_test**: uint8 array of RGB image data with shape (nb_samples, 3, 32, 32).
    - **y_train, y_test**: uint8 array of category labels with shape (nb_samples,).
- **Arguments:**
  - **label_mode**: "fine" or "coarse".

IMDB Movie reviews sentiment classification

Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a sequence of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.

Usage:

```
from keras.datasets import imdb

(X_train, y_train), (X_test, y_test) = imdb.load_data(path="imdb.pkl",
                                                      nb_words=None,
                                                      skip_top=0,
                                                      maxlen=None,
                                                      test_split=0.1)
```

- **Return:**
  - 2 tuples:
    - **X_train, X_test**: list of sequences, which are lists of indexes (integers). If the nb_words argument was specific, the maximum possible index value is nb_words-1. If the maxlen argument was specified, the largest possible sequence length is maxlen.
    - **y_train, y_test**: list of integer labels (1 or 0).
- **Arguments:**
  - **path**: if you do have the data locally (at `'~/.keras/datasets/' + path`), if will be downloaded to this location (in cPickle format).
  - **nb_words**: integer or None. Top most frequent words to consider. Any less frequent word will appear as 0 in the sequence data.
  - **skip_top**: integer. Top most frequent words to ignore (they will appear as 0s in the sequence data).
  - **maxlen**: int. Maximum sequence length. Any longer sequence will be truncated.
  - **test_split**: float. Fraction of the dataset to be used as test data.
  - **seed**: int. Seed for reproducible data shuffling.

Reuters newswire topics classification

Dataset of 11,228 newswires from Reuters, labeled over 46 topics. As with the IMDB dataset, each wire is encoded as a sequence of word indexes (same conventions).

Usage:

```
from keras.datasets import reuters

(X_train, y_train), (X_test, y_test) = reuters.load_data(path="reuters.pkl",
                                                         nb_words=None,
                                                         skip_top=0,
                                                         maxlen=None,
                                                         test_split=0.1)
```

The specifications are the same as that of the IMDB dataset.

This dataset also makes available the word index used for encoding the sequences:

```
word_index = reuters.get_word_index(path="reuters_word_index.pkl")
```

- **Return:** A dictionary where key are words (str) and values are indexes (integer). eg. `word_index["giraffe"]` might return `1234`.
- **Arguments:**
  - **path**: if you do have the index file locally (at `'~/.keras/datasets/' + path`), if will be downloaded to this location (in cPickle format).

MNIST database of handwritten digits

Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.

Usage:

```
from keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

- **Return:**
  - 2 tuples:
    - **X_train, X_test**: uint8 array of grayscale image data with shape (nb_samples, 28, 28).
    - **y_train, y_test**: uint8 array of digit labels (integers in range 0-9) with shape (nb_samples,).
- **Arguments:**
  - **path**: if you do have the index file locally (at `'~/.keras/datasets/' + path`), if will be downloaded to this location (in cPickle format).

Keras backends

What is a "backend"?

Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle itself low-level operations such as tensor products, convolutions and so on. Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.

At this time, Keras has two backend implementations available: the **Theano** backend and the **TensorFlow** backend.

- Theano is an open-source symbolic tensor manipulation framework developed by LISA/MILA Lab at Université de Montréal.
- TensorFlow is an open-source symbolic tensor manipulation framework developed by Google, Inc.

Switching from one backend to another

If you have run Keras at least once, you will find the Keras configuration file at:

`~/.keras/keras.json`

If it isn't there, you can create it.

It probably looks like this:

`{"epsilon": 1e-07, "floatx": "float32", "backend": "theano"}`

Simply change the field `backend` to either `"theano"` or `"tensorflow"`, and Keras will use the new configuration next time you run any Keras code.

You can also define the environment variable `KERAS_BACKEND` and this will override what is defined in your config file :

```
KERAS_BACKEND=tensorflow python -c "from keras import backend; print(backend._BACKEND)"
Using TensorFlow backend.
tensorflow
```

Using the abstract Keras backend to write new code

If you want the Keras modules you write to be compatible with both Theano and TensorFlow, you have to write them via the abstract Keras backend API. Here's an intro.

You can import the backend module via:

```
from keras import backend as K
```

The code below instantiates an input placeholder. It's equivalent to `tf.placeholder()` or `T.matrix()`, `T.tensor3()`, etc.

```
input = K.placeholder(shape=(2, 4, 5))
# also works:
input = K.placeholder(shape=(None, 4, 5))
# also works:
input = K.placeholder(ndim=3)
```

The code below instantiates a shared variable. It's equivalent to `tf.variable()` or `theano.shared()`.

```
val = np.random.random((3, 4, 5))
var = K.variable(value=val)

# all-zeros variable:
var = K.zeros(shape=(3, 4, 5))
# all-ones:
var = K.ones(shape=(3, 4, 5))
```

Most tensor operations you will need can be done as you would in TensorFlow or Theano:

```
a = b + c * K.abs(d)
c = K.dot(a, K.transpose(b))
a = K.sum(b, axis=2)
a = K.softmax(b)
a = concatenate([b, c], axis=-1)
# etc...
```

Backend functions

epsilon

```
epsilon()
```

Returns the value of the fuzz factor used in numeric expressions.

---

set_epsilon

```
set_epsilon(e)
```

Sets the value of the fuzz factor used in numeric expressions.

---

floatx

```
floatx()
```

Returns the default float type, as a string (e.g. 'float16', 'float32', 'float64').

---

cast_to_floatx

```
cast_to_floatx(x)
```

Cast a Numpy array to floatx.

---

image_dim_ordering

```
image_dim_ordering()
```

Returns the image dimension ordering convention ('th' or 'tf').

---

set_image_dim_ordering

```
set_image_dim_ordering(dim_ordering)
```

Sets the value of the image dimension ordering convention ('th' or 'tf').

---

learning_phase

```
learning_phase()
```

Returns the learning phase flag.

The learning phase flag is an integer tensor (0 = test, 1 = train) to be passed as input to any Keras function that uses a different behavior at train time and test time.

---

manual_variable_initialization

```
manual_variable_initialization(value)
```

Whether variables should be initialized as they are instantiated (default), or if the user should handle the initialization (e.g. via tf.initialize_all_variables()).

---

variable

```
variable(value, dtype='float32', name=None)
```

Instantiates a tensor.

Arguments

- **value**: numpy array, initial value of the tensor.
- **dtype**: tensor type.
- **name**: optional name string for the tensor.

Returns

Tensor variable instance.

---

placeholder

```
placeholder(shape=None, ndim=None, dtype='float32', name=None)
```

Instantiates a placeholder.

Arguments

- **shape**: shape of the placeholder (integer tuple, may include None entries).
- **ndim**: number of axes of the tensor. At least one of {`shape`, `ndim`} must be specified. If both are specified, `shape` is used.
- **dtype**: placeholder type.

- **name**: optional name string for the placeholder.

Returns

Placeholder tensor instance.

---

shape

```
shape(x)
```

Returns the symbolic shape of a tensor.

---

int_shape

```
int_shape(x)
```

Returns the shape of a tensor as a tuple of integers or None entries. Note that this function only works with TensorFlow.

---

ndim

```
ndim(x)
```

Returns the number of axes in a tensor, as an integer.

---

dtype

```
dtype(x)
```

Returns the dtype of a tensor, as a string.

---

eval

```
eval(x)
```

Evaluates the value of a tensor. Returns a Numpy array.

---

zeros

```
zeros(shape, dtype='float32', name=None)
```

Instantiates an all-zeros tensor variable.

---

ones

```
ones(shape, dtype='float32', name=None)
```

Instantiates an all-ones tensor variable.

---

eye

```
eye(size, dtype='float32', name=None)
```

Instantiate an identity matrix.

---

zeros_like

```
zeros_like(x, name=None)
```

Instantiates an all-zeros tensor of the same shape as another tensor.

---

ones_like

```
ones_like(x, name=None)
```

Instantiates an all-ones tensor of the same shape as another tensor.

---

count_params

```
count_params(x)
```

Returns the number of scalars in a tensor.

---

cast

```
cast(x, dtype)
```

Casts a tensor to a different dtype.

---

dot

```
dot(x, y)
```

Multiplies 2 tensors. When attempting to multiply a ND tensor with a ND tensor, reproduces the Theano behavior (e.g. (2, 3).(4, 3, 5) = (2, 4, 5))

---

batch_dot

```
batch_dot(x, y, axes=None)
```

Batchwise dot product.

batch_dot results in a tensor with less dimensions than the input. If the number of dimensions is reduced to 1, we use `expand_dims` to make sure that ndim is at least 2.

Arguments

x, y: tensors with ndim >= 2 - **axes**: list (or single) int with target dimensions

Returns

A tensor with shape equal to the concatenation of x's shape (less the dimension that was summed over) and y's shape (less the batch dimension and the dimension that was summed over). If the final rank is 1, we reshape it to (batch_size, 1).

Examples

Assume x = [[1, 2], [3, 4]] and y = [[5, 6], [7, 8]] batch_dot(x, y, axes=1) = [[17, 53]] which is the main diagonal of x.dot(y.T), although we never have to calculate the off-diagonal elements.

Shape inference: Let x's shape be (100, 20) and y's shape be (100, 30, 20). If dot_axes is (1, 2), to find the output shape of resultant tensor, loop through each dimension in x's shape and y's shape: x.shape[0] : 100 : append to output shape x.shape[1] : 20 : do not append to output shape, dimension 1 of x has been summed over. (dot_axes[0] = 1) y.shape[0] : 100 : do not append to output shape, always ignore first dimension of y y.shape[1] : 30 : append to output shape y.shape[2] : 20 : do not append to output shape, dimension 2 of y has been summed over. (dot_axes[1] = 2)

output_shape = (100, 30)

transpose

```
transpose(x)
```

Transposes a matrix.

---

gather

```
gather(reference, indices)
```

Retrieves the vectors of indices `indices` in the 2D tensor `reference`.

Arguments

- **reference**: a 2D tensor.
- **indices**: an int tensor of indices.

Returns

A 3D tensor of same type as `reference`.

---

max

```
max(x, axis=None, keepdims=False)
```

Maximum value in a tensor.

---

min

```
min(x, axis=None, keepdims=False)
```

Minimum value in a tensor.

---

sum

```
sum(x, axis=None, keepdims=False)
```

Sum of the values in a tensor, alongside the specified axis.

---

prod

```
prod(x, axis=None, keepdims=False)
```

Multiplies the values in a tensor, alongside the specified axis.

---

var

```
var(x, axis=None, keepdims=False)
```

Variance of a tensor, alongside the specified axis.

---

std

```
std(x, axis=None, keepdims=False)
```

Standard deviation of a tensor, alongside the specified axis.

---

mean

```
mean(x, axis=None, keepdims=False)
```

Mean of a tensor, alongside the specified axis.

---

any

```
any(x, axis=None, keepdims=False)
```

Bitwise reduction (logical OR).

Returns an uint8 tensor (0s and 1s).

---

all

```
all(x, axis=None, keepdims=False)
```

Bitwise reduction (logical AND).

Returns an uint8 tensor

argmax

```
argmax(x, axis=-1)
```

Returns the index of the maximum value along a tensor axis.

argmin

```
argmin(x, axis=-1)
```

Returns the index of the minimum value along a tensor axis.

square

```
square(x)
```

Element-wise square.

abs

```
abs(x)
```

Element-wise absolute value.

sqrt

```
sqrt(x)
```

Element-wise square root.

exp

```
exp(x)
```

Element-wise exponential.

log

```
log(x)
```

Element-wise log.

---

round

```
round(x)
```

Element-wise rounding to the closest integer.

---

sign

```
sign(x)
```

Element-wise sign.

---

pow

```
pow(x, a)
```

Element-wise exponentiation.

---

clip

```
clip(x, min_value, max_value)
```

Element-wise value clipping.

---

equal

```
equal(x, y)
```

Element-wise equality between two tensors. Returns a bool tensor.

---

not_equal

```
not_equal(x, y)
```

Element-wise inequality between two tensors. Returns a bool tensor.

---

greater

```
greater(x, y)
```

Element-wise truth value of (x > y). Returns a bool tensor.

---

greater_equal

```
greater_equal(x, y)
```

Element-wise truth value of (x >= y). Returns a bool tensor.

---

lesser

```
lesser(x, y)
```

Element-wise truth value of (x < y). Returns a bool tensor.

---

lesser_equal

```
lesser_equal(x, y)
```

Element-wise truth value of (x <= y). Returns a bool tensor.

---

maximum

```
maximum(x, y)
```

Element-wise maximum of two tensors.

---

minimum

```
minimum(x, y)
```

Element-wise minimum of two tensors.

---

sin

```
sin(x)
```

Computes sin of x element-wise.

---

cos

```
cos(x)
```

Computes cos of x element-wise.

---

normalize_batch_in_training

```
normalize_batch_in_training(x, gamma, beta, reduction_axes, epsilon=0.0001)
```

Compute mean and std for batch then apply batch_normalization on batch.

---

batch_normalization

```
batch_normalization(x, mean, std, beta, gamma, epsilon=0.0001)
```

Apply batch normalization on x given mean, std, beta and gamma.

---

concatenate

```
concatenate(tensors, axis=-1)
```

Concantes a list of tensors alongside the specified axis.

---

reshape

```
reshape(x, shape)
```

Reshapes a tensor to the specified shape.

---

permute_dimensions

```
permute_dimensions(x, pattern)
```

Permutes axes in a tensor.

Arguments

- **pattern**: should be a tuple of dimension indices, e.g. (0, 2, 1).

---

resize_images

```
resize_images(X, height_factor, width_factor, dim_ordering)
```

Resizes the images contained in a 4D tensor of shape - [batch, channels, height, width] (for 'th' dim_ordering) - [batch, height, width, channels] (for 'tf' dim_ordering) by a factor of (height_factor, width_factor). Both factors should be positive integers.

---

resize_volumes

```
resize_volumes(X, depth_factor, height_factor, width_factor, dim_ordering)
```

Resize the volume contained in a 5D tensor of shape - [batch, channels, depth, height, width] (for 'th' dim_ordering) - [batch, depth, height, width, channels] (for 'tf' dim_ordering) by a factor of (depth_factor, height_factor, width_factor). All three factors should be positive integers.

---

repeat_elements

```
repeat_elements(x, rep, axis)
```

Repeats the elements of a tensor along an axis, like np.repeat

If x has shape (s1, s2, s3) and axis=1, the output will have shape (s1, s2 * rep, s3)

---

repeat

```
repeat(x, n)
```

Repeats a 2D tensor:

if x has shape (samples, dim) and n=2, the output will have shape (samples, 2, dim)

---

batch_flatten

```
batch_flatten(x)
```

Turn a n-D tensor into a 2D tensor where the first dimension is conserved.

---

expand_dims

```
expand_dims(x, dim=-1)
```

Adds a 1-sized dimension at index "dim".

---

squeeze

```
squeeze(x, axis)
```

Removes a 1-dimension from the tensor at index "axis".

---

temporal_padding

```
temporal_padding(x, padding=1)
```

Pads the middle dimension of a 3D tensor with "padding" zeros left and right.

---

spatial_2d_padding

```
spatial_2d_padding(x, padding=(1, 1), dim_ordering='th')
```

Pads the 2nd and 3rd dimensions of a 4D tensor with "padding[0]" and "padding[1]" (resp.) zeros left and right.

---

spatial_3d_padding

```
spatial_3d_padding(x, padding=(1, 1, 1), dim_ordering='th')
```

Pads 5D tensor with zeros for the depth, height, width dimension with "padding[0]", "padding[1]" and "padding[2]" (resp.) zeros left and right

For 'tf' dim_ordering, the 2nd, 3rd and 4th dimension will be padded. For 'th' dim_ordering, the 3rd, 4th and 5th dimension will be padded.

---

get_value

```
get_value(x)
```

Returns the value of a tensor variable, as a Numpy array.

---

batch_get_value

```
batch_get_value(xs)
```

Returns the value of more than one tensor variable, as a list of Numpy arrays.

---

set_value

```
set_value(x, value)
```

Sets the value of a tensor variable, from a Numpy array.

---

batch_set_value

```
batch_set_value(tuples)
```

Sets the values of many tensor variables at once.

Arguments

- **tuples**: a list of tuples `(tensor, value)`. `value` should be a Numpy array.

---

print_tensor

```
print_tensor(x, message='')
```

Print the message and the tensor when evaluated and return the same tensor.

---

function

```
function(inputs, outputs, updates=[])
```

Instantiates a Keras function.

Arguments

- **inputs**: list of placeholder/variable tensors.
- **outputs**: list of output tensors.
- **updates**: list of update tuples (old_tensor, new_tensor).

---

gradients

```
gradients(loss, variables)
```

Returns the gradients of `variables` (list of tensor variables) with regard to `loss`.

---

stop_gradient

```
stop_gradient(variables)
```

Returns `variables` but with zero gradient with respect to every other variables.

---

rnn

```
rnn(step_function, inputs, initial_states, go_backwards=False, mask=None, constants=None, unrol
```

Iterates over the time dimension of a tensor.

Arguments

- **inputs**: tensor of temporal data of shape (samples, time, ...) (at least 3D).
- **step_function**:
- **Parameters**:
    - **input**: tensor with shape (samples, ...) (no time dimension), representing input for the batch of samples at a certain time step.
    - **states**: list of tensors.
- **Returns**:
    - **output**: tensor with shape (samples, ...) (no time dimension),
    - **new_states**: list of tensors, same length and shapes as 'states'.

- **initial_states**: tensor with shape (samples, ...) (no time dimension), containing the initial values for the states used in the step function.
- **go_backwards**: boolean. If True, do the iteration over the time dimension in reverse order.
- **mask**: binary tensor with shape (samples, time, 1), with a zero for every element that is masked.
- **constants**: a list of constant values passed at each step.
- **unroll**: with TensorFlow the RNN is always unrolled, but with Theano you can use this boolean flag to unroll the RNN.
- **input_length**: not relevant in the TensorFlow implementation. Must be specified if using unrolling with Theano.

Returns

A tuple (last_output, outputs, new_states).

- **last_output**: the latest output of the rnn, of shape (samples, ...)
- **outputs**: tensor with shape (samples, time, ...) where each entry outputs[s, t] is the output of the step function at time t for sample s.
- **new_states**: list of tensors, latest states returned by the step function, of shape (samples, ...).

---

switch

```
switch(condition, then_expression, else_expression)
```

Switches between two operations depending on a scalar value (int or bool). Note that both `then_expression` and `else_expression` should be symbolic tensors of the *same shape*.

Arguments

- **condition**: scalar tensor.
- **then_expression**: TensorFlow operation.
- **else_expression**: TensorFlow operation.

---

in_train_phase

```
in_train_phase(x, alt)
```

Selects `x` in train phase, and `alt` otherwise. Note that `alt` should have the *same shape* as `x`.

---

in_test_phase

```
in_test_phase(x, alt)
```

Selects `x` in test phase, and `alt` otherwise. Note that `alt` should have the *same shape* as `x`.

---

relu

```
relu(x, alpha=0.0, max_value=None)
```

Rectified linear unit

Arguments

- **alpha**: slope of negative section.
- **max_value**: saturation threshold.

---

softmax

```
softmax(x)
```

Softmax of a tensor.

---

softplus

```
softplus(x)
```

Softplus of a tensor.

---

categorical_crossentropy

```
categorical_crossentropy(output, target, from_logits=False)
```

Categorical crossentropy between an output tensor and a target tensor, where the target is a tensor of the same shape as the output.

---

sparse_categorical_crossentropy

```
sparse_categorical_crossentropy(output, target, from_logits=False)
```

Categorical crossentropy between an output tensor and a target tensor, where the target is an integer tensor.

---

binary_crossentropy

```
binary_crossentropy(output, target, from_logits=False)
```

Binary crossentropy between an output tensor and a target tensor.

---

sigmoid

```
sigmoid(x)
```

Element-wise sigmoid.

---

hard_sigmoid

```
hard_sigmoid(x)
```

Segment-wise linear approximation of sigmoid. Faster than sigmoid.

---

tanh

```
tanh(x)
```

Element-wise tanh.

---

dropout

```
dropout(x, level, seed=None)
```

Sets entries in `x` to zero at random, while scaling the entire tensor.

Arguments

- **x**: tensor
- **level**: fraction of the entries in the tensor that will be set to 0
- **seed**: random seed to ensure determinism.

---

l2_normalize

```
l2_normalize(x, axis)
```

Normalizes a tensor wrt the L2 norm alongside the specified axis.

---

conv2d

```
conv2d(x, kernel, strides=(1, 1), border_mode='valid', dim_ordering='th', image_shape=None, fil
```

2D convolution.

Arguments

- **kernel**: kernel tensor.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

deconv2d

```
deconv2d(x, kernel, output_shape, strides=(1, 1), border_mode='valid', dim_ordering='th', image
```

2D deconvolution (i.e. transposed convolution).

Arguments

- **x**: input tensor.
- **kernel**: kernel tensor.
- **output_shape**: 1D int tensor for the output shape.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

conv3d

```
conv3d(x, kernel, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', volume_shape=None,
```

3D convolution.

Arguments

- **kernel**: kernel tensor.
- **strides**: strides tuple.
- **border_mode**: string, "same" or "valid".
- **dim_ordering**: "tf" or "th". Whether to use Theano or TensorFlow dimension ordering for inputs/kernels/ouputs.

---

pool2d

```
pool2d(x, pool_size, strides=(1, 1), border_mode='valid', dim_ordering='th', pool_mode='max')
```

2D Pooling.

Arguments

- **pool_size**: tuple of 2 integers.
- **strides**: tuple of 2 integers.
- **border_mode**: one of "valid", "same".
- **dim_ordering**: one of "th", "tf".
- **pool_mode**: one of "max", "avg".

---

pool3d

```
pool3d(x, pool_size, strides=(1, 1, 1), border_mode='valid', dim_ordering='th', pool_mode='max'
```

3D Pooling.

Arguments

- **pool_size**: tuple of 3 integers.
- **strides**: tuple of 3 integers.
- **border_mode**: one of "valid", "same".
- **dim_ordering**: one of "th", "tf".
- **pool_mode**: one of "max", "avg".

Usage of initializations

Initializations define the way to set the initial random weights of Keras layers.

The keyword arguments used for passing initializations to layers will depend on the layer. Usually it is simply `init`:

```
model.add(Dense(64, init='uniform'))
```

Available initializations

- **uniform**
- **lecun_uniform**: Uniform initialization scaled by the square root of the number of inputs (LeCun 98).
- **normal**
- **identity**: Use with square 2D layers (`shape[0] == shape[1]`).
- **orthogonal**: Use with square 2D layers (`shape[0] == shape[1]`).
- **zero**
- **glorot_normal**: Gaussian initialization scaled by fan_in + fan_out (Glorot 2010)
- **glorot_uniform**
- **he_normal**: Gaussian initialization scaled by fan_in (He et al., 2014)
- **he_uniform**

An initialization may be passed as a string (must match one of the available initializations above), or as a callable. If a callable, then it must take two arguments: `shape` (shape of the variable to initialize) and `name` (name of the variable), and it must return a variable (e.g. output of `K.variable()`):

```
from keras import backend as K
import numpy as np

def my_init(shape, name=None):
    value = np.random.random(shape)
    return K.variable(value, name=name)

model.add(Dense(64, init=my_init))
```

You could also use functions from `keras.initializations` in this way:

```
from keras import initializations

def my_init(shape, name=None):
    return initializations.normal(shape, scale=0.01, name=name)

model.add(Dense(64, init=my_init))
```

Usage of regularizers

Regularizers allow to apply penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes.

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `TimeDistributedDense`, `MaxoutDense`, `Convolution1D` and `Convolution2D` have a unified API.

These layers expose 3 keyword arguments:

- `W_regularizer`: instance of `keras.regularizers.WeightRegularizer`
- `b_regularizer`: instance of `keras.regularizers.WeightRegularizer`
- `activity_regularizer`: instance of `keras.regularizers.ActivityRegularizer`

Example

```
from keras.regularizers import l2, activity_l2
model.add(Dense(64, input_dim=64, W_regularizer=l2(0.01), activity_regularizer=activity_l2(0.01
```

Available penalties

```
keras.regularizers.WeightRegularizer(l1=0., l2=0.)
```

```
keras.regularizers.ActivityRegularizer(l1=0., l2=0.)
```

Shortcuts

These are shortcut functions available in `keras.regularizers`.

- **l1**(l=0.01): L1 weight regularization penalty, also known as LASSO
- **l2**(l=0.01): L2 weight regularization penalty, also known as weight decay, or Ridge
- **l1l2**(l1=0.01, l2=0.01): L1-L2 weight regularization penalty, also known as ElasticNet
- **activity_l1**(l=0.01): L1 activity regularization
- **activity_l2**(l=0.01): L2 activity regularization
- **activity_l1l2**(l1=0.01, l2=0.01): L1+L2 activity regularization

Usage of constraints

Functions from the `constraints` module allow setting constraints (eg. non-negativity) on network parameters during optimization.

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `TimeDistributedDense`, `MaxoutDense`, `Convolution1D` and `Convolution2D` have a unified API.

These layers expose 2 keyword arguments:

- `W_constraint` for the main weights matrix
- `b_constraint` for the bias.

```
from keras.constraints import maxnorm
model.add(Dense(64, W_constraint = maxnorm(2)))
```

Available constraints

- **maxnorm**(m=2): maximum-norm constraint
- **nonneg**(): non-negativity constraint
- **unitnorm**(): unit-norm constraint, enforces the matrix to have unit norm along the last axis

Model visualization

The `keras.utils.visualize_util` module provides utility functions to plot a Keras model (using graphviz).

This will plot a graph of the model and save it to a file:

```
from keras.utils.visualize_util import plot
plot(model, to_file='model.png')
```

`plot` takes two optional arguments:

- `show_shapes` (defaults to False) controls whether output shapes are shown in the graph.
- `show_layer_names` (defaults to True) controls whether layer names are shown in the graph.

You can also directly obtain the `pydot.Graph` object and render it yourself, for example to show it in an ipython notebook :

```
from IPython.display import SVG
from keras.utils.visualize_util import model_to_dot

SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Wrappers for the Scikit-Learn API

You can use `Sequential` Keras models (single-input only) as part of your Scikit-Learn workflow via the wrappers found at `keras.wrappers.sklearn.py`.

There are two wrappers available:

`keras.wrappers.sklearn.KerasClassifier(build_fn=None, **sk_params)`, which implements the sklearn classifier interface,

`keras.wrappers.sklearn.KerasRegressor(build_fn=None, **sk_params)`, which implements the sklearn regressor interface.

Arguments

- **build_fn**: callable function or class instance
- **sk_params**: model parameters & fitting parameters

`build_fn` should construct, compile and return a Keras model, which will then be used to fit/predict. One of the following three values could be passed to build_fn:

1. A function
2. An instance of a class that implements the **call** method
3. None. This means you implement a class that inherits from either `KerasClassifier` or `KerasRegressor`. The **call** method of the present class will then be treated as the default build_fn.

`sk_params` takes both model parameters and fitting parameters. Legal model parameters are the arguments of `build_fn`. Note that like all other estimators in scikit-learn, 'build_fn' should provide default values for its arguments, so that you could create the estimator without passing any values to `sk_params`.

`sk_params` could also accept parameters for calling `fit`, `predict`, `predict_proba`, and `score` methods (e.g., `nb_epoch`, `batch_size`). fitting (predicting) parameters are selected in the following order:

1. Values passed to the dictionary arguments of `fit`, `predict`, `predict_proba`, and `score` methods
2. Values passed to `sk_params`

3. The default values of the `keras.models.Sequential` `fit`, `predict`, `predict_proba` and `score` methods

When using scikit-learn's `grid_search` API, legal tunable parameters are those you could pass to `sk_params`, including fitting parameters. In other words, you could use `grid_search` to search for the best `batch_size` or `nb_epoch` as well as the model parameters.