# Home

## Keras: Deep Learning library for Theano and TensorFlow

## You have just found Keras.

Keras is a high-level neural networks library, written in Python and capable of running on top of either [TensorFlow](https://github.com/tensorflow/tensorflow) or [Theano](https://github.com/Theano/Theano). It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Use Keras if you need a deep learning library that:

* Allows for easy and fast prototyping (through total modularity, minimalism, and extensibility).
* Supports both convolutional networks and recurrent networks, as well as combinations of the two.
* Supports arbitrary connectivity schemes (including multi-input and multi-output training).
* Runs seamlessly on CPU and GPU.

Read the documentation at [Keras.io](http://keras.io/).

Keras is compatible with: **Python 2.7-3.5**.

## Guiding principles

* **Modularity.** A model is understood as a sequence or a graph of standalone, fully-configurable modules that can be plugged together with as little restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, regularization schemes are all standalone modules that you can combine to create new models.
* **Minimalism.** Each module should be kept short and simple. Every piece of code should be transparent upon first reading. No black magic: it hurts iteration speed and ability to innovate.
* **Easy extensibility.** New modules are dead simple to add (as new classes and functions), and existing modules provide ample examples. To be able to easily create new modules allows for total expressiveness, making Keras suitable for advanced research.
* **Work with Python**. No separate models configuration files in a declarative format. Models are described in Python code, which is compact, easier to debug, and allows for ease of extensibility.

## Getting started: 30 seconds to Keras

The core data structure of Keras is a **model**, a way to organize layers. The main type of model is the [Sequential](http://keras.io/getting-started/sequential-model-guide)model, a linear stack of layers. For more complex architectures, you should use the [Keras functional API](http://keras.io/getting-started/functional-api-guide).

Here's the Sequential model:

**from** keras.models **import** Sequential

model = Sequential()

Stacking layers is as easy as .add():

**from** keras.layers **import** Dense, Activation

model.add(Dense(output\_dim=64, input\_dim=100))

model.add(Activation("relu"))

model.add(Dense(output\_dim=10))

model.add(Activation("softmax"))

Once your model looks good, configure its learning process with .compile():

model.compile(loss='categorical\_crossentropy', optimizer='sgd', metrics=['accuracy'])

If you need to, you can further configure your optimizer. A core principle of Keras is to make things reasonably simple, while allowing the user to be fully in control when they need to (the ultimate control being the easy extensibility of the source code).

**from** keras.optimizers **import** SGD

model.compile(loss='categorical\_crossentropy', optimizer=SGD(lr=0.01, momentum=0.9, nesterov=**True**))

You can now iterate on your training data in batches:

model.fit(X\_train, Y\_train, nb\_epoch=5, batch\_size=32)

Alternatively, you can feed batches to your model manually:

model.train\_on\_batch(X\_batch, Y\_batch)

Evaluate your performance in one line:

loss\_and\_metrics = model.evaluate(X\_test, Y\_test, batch\_size=32)

Or generate predictions on new data:

classes = model.predict\_classes(X\_test, batch\_size=32)

proba = model.predict\_proba(X\_test, batch\_size=32)

Building a question answering system, an image classification model, a Neural Turing Machine, a word2vec embedder or any other model is just as fast. The ideas behind deep learning are simple, so why should their implementation be painful?

For a more in-depth tutorial about Keras, you can check out:

* [Getting started with the Sequential model](http://keras.io/getting-started/sequential-model-guide)
* [Getting started with the functional API](http://keras.io/getting-started/functional-api-guide)

In the [examples folder](https://github.com/fchollet/keras/tree/master/examples) of the repository, you will find more advanced models: question-answering with memory networks, text generation with stacked LSTMs, etc.

## Installation

Keras uses the following dependencies:

* numpy, scipy
* pyyaml
* HDF5 and h5py (optional, required if you use model saving/loading functions)
* Optional but recommended if you use CNNs: cuDNN.

When using the TensorFlow backend:

* TensorFlow
  + [See installation instructions](https://github.com/tensorflow/tensorflow#download-and-setup).

When using the Theano backend:

* Theano
  + [See installation instructions](http://deeplearning.net/software/theano/install.html#install).

To install Keras, cd to the Keras folder and run the install command:

sudo python setup.py install

You can also install Keras from PyPI:

sudo pip install keras

## Switching from TensorFlow to Theano

By default, Keras will use TensorFlow as its tensor manipulation library. [Follow these instructions](http://keras.io/backend/) to configure the Keras backend.

## Support

You can ask questions and join the development discussion:

* On the [Keras Google group](https://groups.google.com/forum/#!forum/keras-users).
* On the [Keras Slack channel](https://kerasteam.slack.com/). Use [this link](https://keras-slack-autojoin.herokuapp.com/) to request an invitation to the channel.

You can also post **bug reports and feature requests** (only) in [Github issues](https://github.com/fchollet/keras/issues). Make sure to read [our guidelines](https://github.com/fchollet/keras/blob/master/CONTRIBUTING.md) first.

## Why this name, Keras?

Keras (κέρας) means horn in Greek. It is a reference to a literary image from ancient Greek and Latin literature, first found in the Odyssey, where dream spirits (Oneiroi, singular Oneiros) are divided between those who deceive men with false visions, who arrive to Earth through a gate of ivory, and those who announce a future that will come to pass, who arrive through a gate of horn. It's a play on the words κέρας (horn) / κραίνω (fulfill), and ἐλέφας (ivory) / ἐλεφαίρομαι (deceive).

Keras was initially developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).

"Oneiroi are beyond our unravelling --who can be sure what tale they tell? Not all that men look for comes to pass. Two gates there are that give passage to fleeting Oneiroi; one is made of horn, one of ivory. The Oneiroi that pass through sawn ivory are deceitful, bearing a message that will not be fulfilled; those that come out through polished horn have truth behind them, to be accomplished for men who see them." Homer, Odyssey 19. 562 ff (Shewring translation).

# Getting started

## 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 Noneindicates 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()

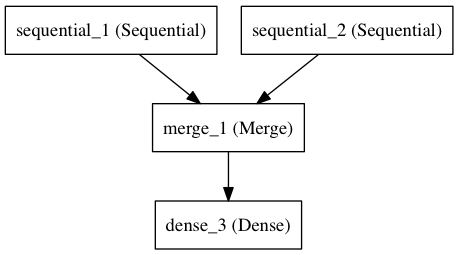
right\_branch.add(Dense(32, input\_dim=784))

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 input*

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: x[0] - x[1])

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](https://keras.io/getting-started/functional-api-guide).

## 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](https://keras.io/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](https://keras.io/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 or a custom metric function. Custom metric function should return either a single tensor value or a dict metric\_name -> metric\_value. See: [metrics](https://keras.io/metrics).

*# 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')

*# for custom metrics*

**import** keras.backend **as** K

**def** **mean\_pred**(y\_true, y\_pred):

**return** K.mean(y\_pred)

**def** **false\_rates**(y\_true, y\_pred):

false\_neg = ...

false\_pos = ...

**return** {

'false\_neg': false\_neg,

'false\_pos': false\_pos,

}

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy', mean\_pred, false\_rates])

## 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](https://keras.io/models/sequential).

*# 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:

**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:

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:

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:

**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:

**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*

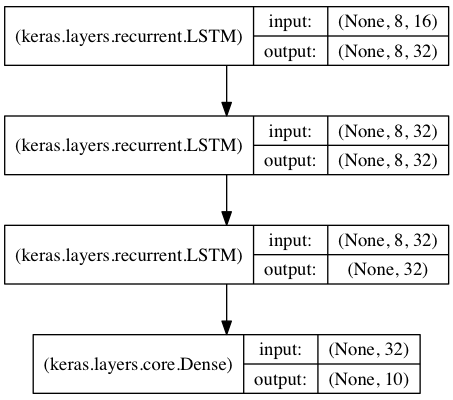
*# partial caption.*

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 dimension 32*

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.](https://keras.io/faq/#how-can-i-use-stateful-rnns)

**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,

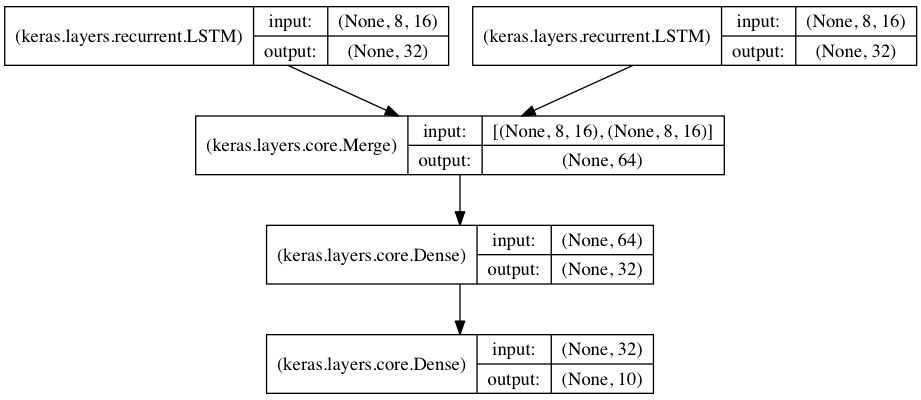
batch\_size=batch\_size, nb\_epoch=5,

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.



**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))