CHAPTER 7

Introduction to Keras

This chapter introduces the reader to Keras, which is a library that provides highly powerful and abstract building blocks to build deep learning networks. The building blocks Keras provides are built using Theano (covered earlier) as well as TensorFlow (which is an alternative to Theano for building computational graphs, automatically deriving gradients, etc.). Keras supports both CPU and GPU computation and is a great tool for quickly prototyping ideas.

We will introduce a number of key building blocks Keras provides, and then build a CNN and LSTM using Keras.

Let us start with a simple, single layer neural network. Listing 7-1 provides the code and Figure 7-1 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct, which allows the user to add/configure layers.
2. Using this functionality, a user can add one or more layers and build the network. The Dense layer is basically a fully connected layer (leading to a vector-matrix or vector-vector product), which we have seen earlier.
3. The input and output dimensionality needs to be specified when the first layer is defined. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 1.
4. After this layer we add an activation function, in this case a sigmoid.
5. The model once defined needs to be explicitly compiled and, at this time, we provide the loss function, the optimization algorithm, and other metrics we want to calculate.
6. An appropriate loss function needs to be picked given the task at hand; in this case, given that we have a binary classification problem, we select binary cross-entropy.
7. An appropriate optimization algorithm needs to be picked, which typically is a variant of Stochastic Gradient Descent (coved in later chapters).
8. Once compiled we can fit the model by providing the data and evaluate the model.

N. Ketkar, *Deep Learning with Python*, DOI 10.1007/978-1-4842-2766-4_7

Listing 7-1. Single Layer Neural Network

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(1, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)
plot(model, to_file='s1.png', show_shapes=True)

# Before Training: [('loss', 0.76832762384414677), ('acc', 0.50700000000000001)]
# After Training: [('loss', 0.67270196056365972), ('acc', 0.56299999999999994)]
```

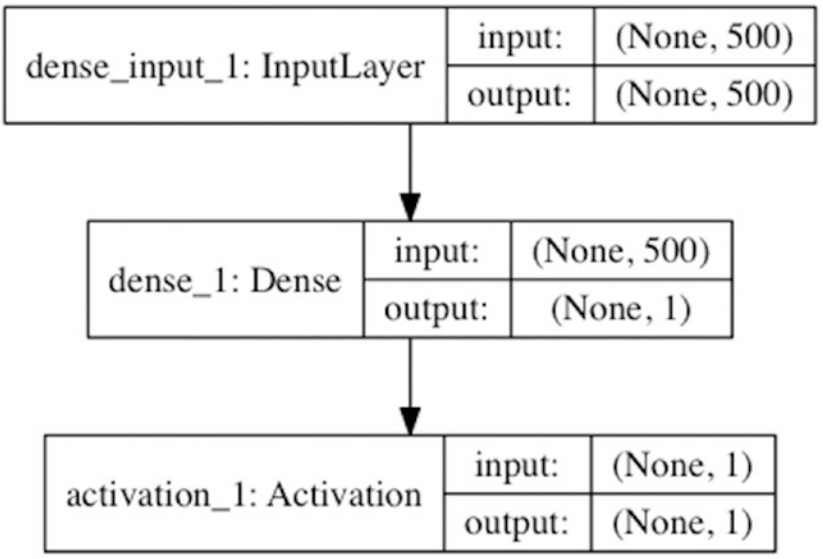

Figure 7-1. *Single Layer Neural Network (Binary Classification)*

Listing 7-2. Two Layer Neural Network

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(32, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.add(Dense(1))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s2.png', show_shapes=True)

# Before Training: [('loss', 0.73012506151199341), ('acc', 0.51200000000000001)]
# After Training: [('loss', 0.6588478517532349), ('acc', 0.52700000000000002)]
```

Let us now look at a two-layer neural network. Listing 7-2 provides the code and Figure 7-2 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense. Here we define the output dimensionality to be 1. Note, however, that we do not need to define the input dimensionality, as it is the same as the dimensionality of the output of the previous layer.
5. As before, we define the optimize and loss function, compile, train, and evaluate.

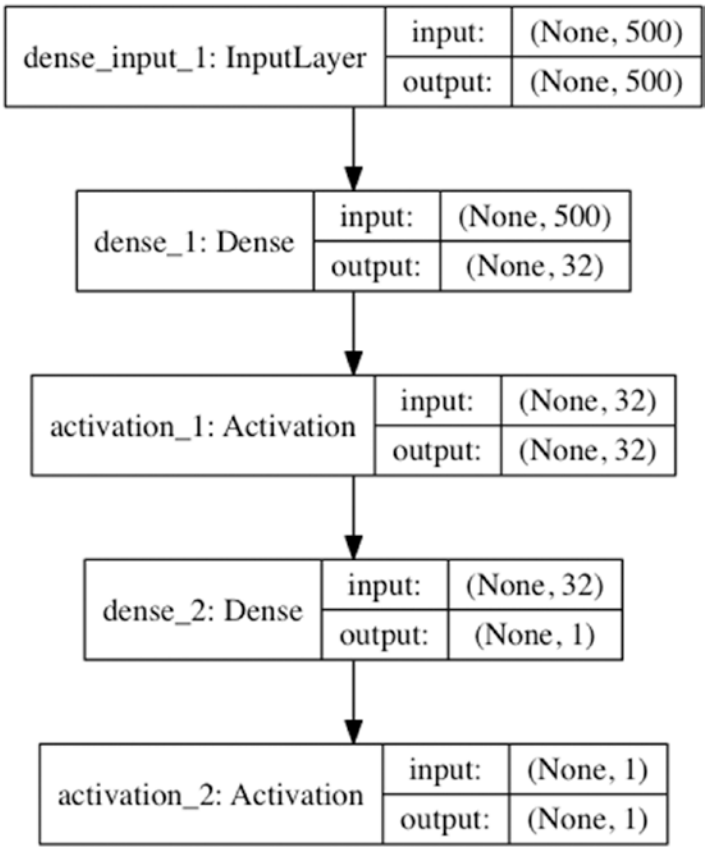

Figure 7-2. *Double Layer Neural Network (Binary Classification)*

Let us now look at a two-layer neural network for multiclass classification. Listing 7-3 provides the code and Figure 7-3 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense. Here we define the output dimensionality to be 10. Note that this is exactly equal to the number of classes we have in our dataset.
5. Next we use the softmax activation and the categorical entropy as the loss function (an earlier chapter covers why this is a good choice).
6. We compile, train, and evaluate the model as before.

Listing 7-3. Multiclass Classification

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.np_utils import to_categorical
from keras.utils.visualize_util import plot

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

data = np.random.random((1000, 500))
labels = to_categorical(np.random.randint(10, size=(1000, 1)))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s3.png', show_shapes=True)

# Before Training: [('loss', 2.4697211952209472), ('categorical_accuracy',
0.092999999999999999)]
# After Training: [('loss', 2.1891849689483642), ('categorical_accuracy',
0.19400000000000001)]
```

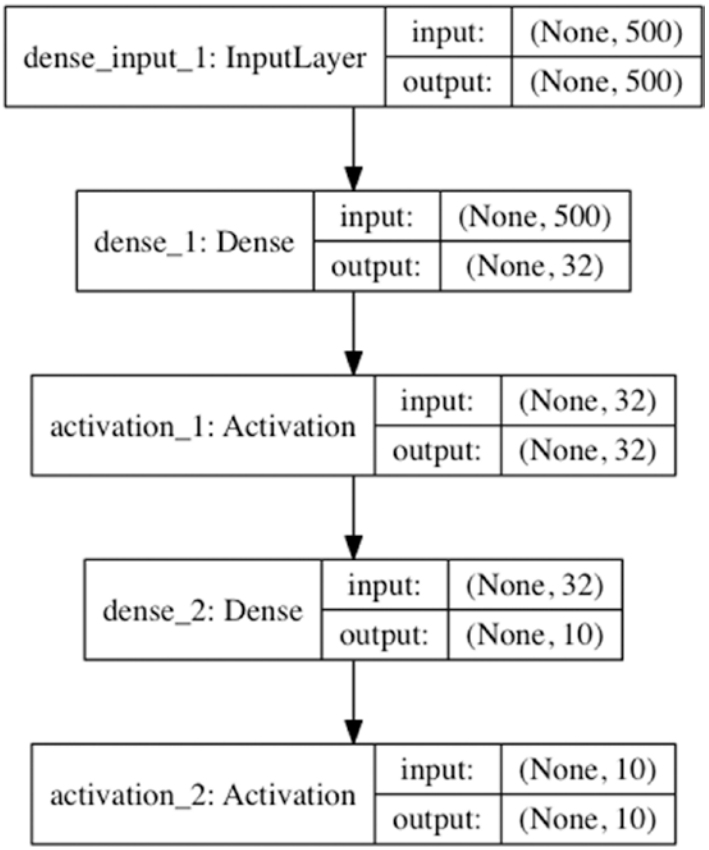

Figure 7-3. *Multiclass Classification*

Let us now look at a two-layer neural network for regression. Listing 7-4 provides the code and Figure 7-4 gives the computational graph. The following points are to be noted:

1. A model is defined using the Sequential construct.
2. We add the first layer, using Dense, and specify the input dimensionality. In this case the model will take an input of dimensionality 500 and produce an output of dimensionality 32.
3. We define an activation function, selecting sigmoid.
4. We then define the second layer using Dense, producing an output of dimensionality 1.
5. We select the activation as sigmoid and select mean squared error, which is appropriate for regression.
6. We compile, train, and evaluate the model.

Listing 7-4. Regression

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils.visualize_util import plot

model = Sequential()
model.add(Dense(32, input_dim=500))
model.add(Activation(activation='sigmoid'))
model.add(Dense(1))
model.add(Activation(activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='mse', metrics=['mean_squared_error'])

data = np.random.random((1000, 500))
labels = np.random.randint(2, size=(1000, 1))

score = model.evaluate(data,labels, verbose=0)
print "Before Training:", zip(model.metrics_names, score)

model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

score = model.evaluate(data,labels, verbose=0)
print "After Training:", zip(model.metrics_names, score)

plot(model, to_file='s4.png', show_shapes=True)

# Before Training: [('loss', 0.26870122766494753), ('mean_squared_error',
0.26870122766494753)]
# After Training: [('loss', 0.22180086207389832), ('mean_squared_error',
0.22180086207389832)]
```

Let us now take a pause and look at how Keras allows for quick iteration of ideas.

1. New models can be quickly defined, trained, and evaluated using the sequential construct.
2. The parameters of the layers input/output dimensionality can be easily modified.
3. We can compare multiple choices of activation functions easily. Listing 7-6 illustrates how we can compare the effects of activation functions.
4. We can compare multiple choices of optimization algorithms easily. Listing 7-5 illustrates how we can compare the effects of different choices of activation algorithms.

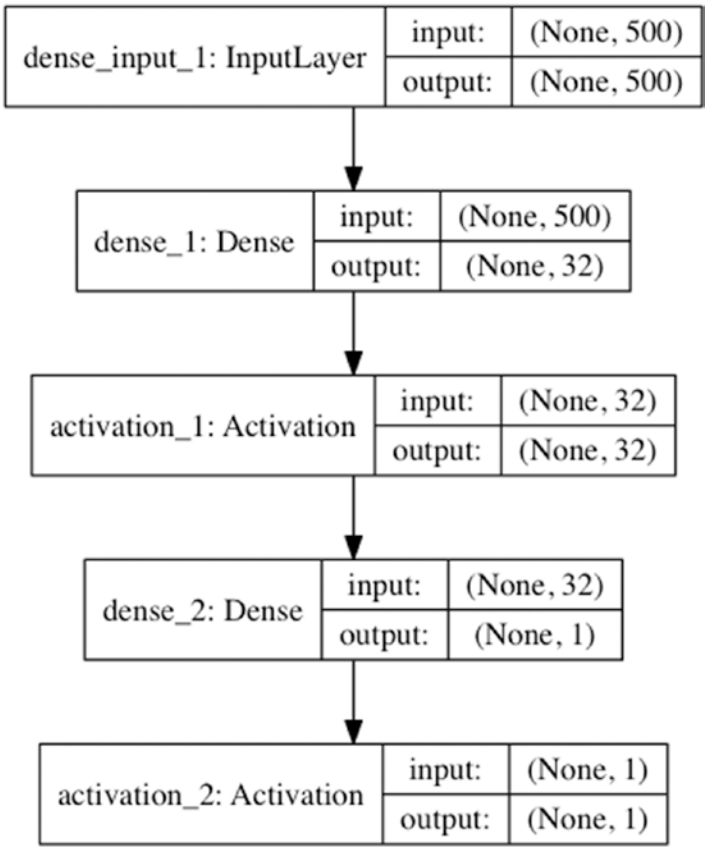

Figure 7-4. *Regression*

Listing 7-5. Optimisers

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

def train_given_optimiser(optimiser):
    model = Sequential()
    model.add(Dense(1, input_dim=500))
    model.add(Activation(activation='sigmoid'))
    model.compile(optimizer=optimiser, loss='binary_crossentropy', metrics=['accuracy'])

    data = np.random.random((1000, 500))
    labels = np.random.randint(2, size=(1000, 1))

    score = model.evaluate(data,labels, verbose=0)
    print "Optimiser: ", optimiser
    print "Before Training:", zip(model.metrics_names, score)
```

```
    model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

    score = model.evaluate(data,labels, verbose=0)
    print "After Training:", zip(model.metrics_names, score)

train_given_optimiser("sgd")
train_given_optimiser("rmsprop")
train_given_optimiser("adagrad")
train_given_optimiser("adadelta")
train_given_optimiser("adam")
train_given_optimiser("adamax")
train_given_optimiser("nadam")

# Optimiser:  sgd
# Before Training: [('loss', 0.76416229248046874), ('acc', 0.51800000000000002)]
# After Training: [('loss', 0.6759231286048889), ('acc', 0.56899999999999995)]
# Optimiser:  rmsprop
# Before Training: [('loss', 0.77773557662963866), ('acc', 0.52600000000000002)]
# After Training: [('loss', 0.727150842666626), ('acc', 0.53500000000000003)]
# Optimiser:  adagrad
# Before Training: [('loss', 0.9275067367553711), ('acc', 0.49099999999999999)]
# After Training: [('loss', 0.66770141410827633), ('acc', 0.57599999999999996)]
# Optimiser:  adadelta
# Before Training: [('loss', 0.76523585319519039), ('acc', 0.48799999999999999)]
# After Training: [('loss', 0.70753741836547857), ('acc', 0.51700000000000002)]
# Optimiser:  adam
# Before Training: [('loss', 0.76974405097961429), ('acc', 0.51100000000000001)]

# After Training: [('loss', 0.66079518222808842), ('acc', 0.59399999999999997)]
# Optimiser:  adamax
# Before Training: [('loss', 0.76244759178161625), ('acc', 0.49399999999999999)]
# After Training: [('loss', 0.67273861455917361), ('acc', 0.58499999999999996)]
# Optimiser:  nadam
# Before Training: [('loss', 0.71690645027160649), ('acc', 0.50600000000000001)]
# After Training: [('loss', 0.62006913089752203), ('acc', 0.68799999999999994)]
```

Keras implements a number of optimisers, namely Stocastic Gradient Descent (SGD), RMSProp, AdaGrad, AdataDelta, Adam, Adamax, and Nadam. Chapter 8 covers these (SGD and its variants) in much detail, explaining the intuition for each. For the context of this chapter it suffices to say that Keras makes it easy for users to experiment with these optimisers with very little coding effort.

Listing 7-6. Activation Functions

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

def train_given_activation(activation):
    model = Sequential()
    model.add(Dense(1, input_dim=500))
    model.add(Activation(activation=activation))
    model.compile(optimizer="sgd", loss='binary_crossentropy', metrics=['accuracy'])
```

```
    data = np.random.random((1000, 500))
    labels = np.random.randint(2, size=(1000, 1))

    score = model.evaluate(data,labels, verbose=0)
    print "Activation: ", activation
    print "Before Training:", zip(model.metrics_names, score)

    model.fit(data, labels, nb_epoch=10, batch_size=32, verbose=0)

    score = model.evaluate(data,labels, verbose=0)
    print "After Training:", zip(model.metrics_names, score)

train_given_activation("relu")
train_given_activation("tanh")
train_given_activation("sigmoid")
train_given_activation("hard_sigmoid")
train_given_activation("linear")

# Activation:  relu
# Before Training: [('loss', 2.6973885402679443), ('acc', 0.48899999999999999)]
# After Training: [('loss', 7.7373054656982418), ('acc', 0.505)]
# Activation:  tanh
# Before Training: [('loss', 5.0640698051452633), ('acc', 0.41699999999999998)]
# After Training: [('loss', 7.6523446731567386), ('acc', 0.52000000000000002)]
# Activation:  sigmoid
# Before Training: [('loss', 0.70816111516952518), ('acc', 0.52500000000000002)]
# After Training: [('loss', 0.67464308834075926), ('acc', 0.58199999999999996)]
# Activation:  hard_sigmoid
# Before Training: [('loss', 0.70220352411270137), ('acc', 0.52100000000000002)]
# After Training: [('loss', 0.67294596910476689), ('acc', 0.58099999999999996)]
# Activation:  linear
# Before Training: [('loss', 3.5439299507141113), ('acc', 0.47799999999999998)]
# After Training: [('loss', 8.2581552581787108), ('acc', 0.0)]
```

Keras implements a number of activation functions, namely, tanh, sigmoid, hard_sigmoid, linear, and relu (rectified linear unit). Activation functions and their appropriateness given a task (classification, multiclassification, regression, etc.) are covered in much detail in Chapter 3. For the context of this chapter, it suffices to say that Keras makes it easy for users to experiment with these activation functions with very little coding effort.

Let us now look at the constructs Keras provides to build the Convolution Neural Networks introduced in Chapter 5. The data set we will be using is the MNIST data set, which is a commonly used benchmark data set for deep learning. The data set consists of handwritten digits (60,000 training examples and 10,000 test examples). The task at hand is to predict the digit given the image, so this is a multiclassifcation problem with ten classes.

Listing 7-7. CNN

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
```

```
from keras.utils import np_utils
from keras.utils.visualize_util import plot

# Image Size
img_rows, img_cols = 28, 28

# Filter
nb_filters = 32

# Pooling
pool_size = (2, 2)

# Kernel
kernel_size = (3, 3)

# Prepare dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
nb_classes = 10
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

# CNN
model = Sequential()
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1], border_mode='valid',
input_shape=input_shape))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

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

# Training
batch_size = 128
nb_epoch = 1
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch, verbose=1, validation_
data=(X_test, Y_test))
```

```
# Evaluation
score = model.evaluate(X_test, Y_test, verbose=0)
print "Test Metrics:", zip(model.metrics_names, score)
plot(model, to_file='s7.png', show_shapes=True)

# Output
# Train on 60000 samples, validate on 10000 samples
# Epoch 1/1
# 60000/60000 [==============================] - 128s - loss: 0.3964 - acc: 0.8776 - val_
loss: 0.0929 - val_acc: 0.9712
# Test Metrics: [('loss', 0.092853568810969594), ('acc', 0.97119999999999995)]
```

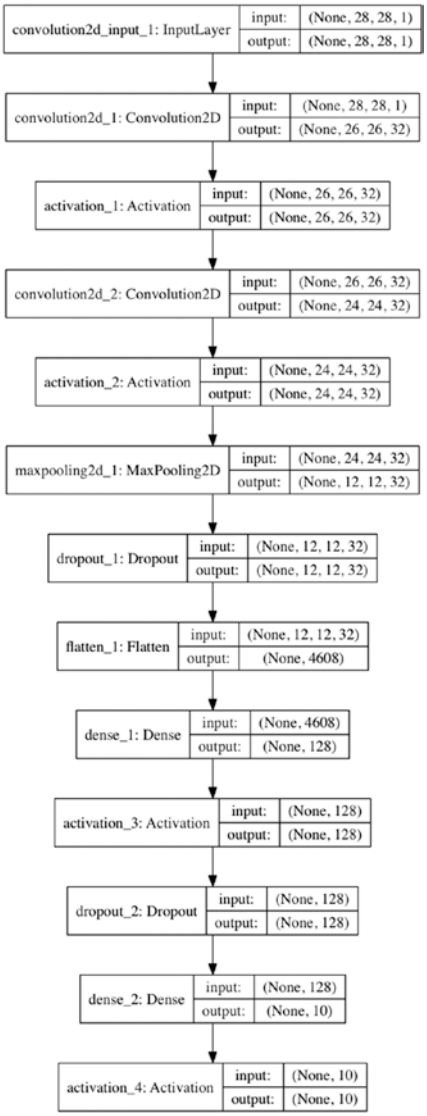

Figure 7-5. *CNN*

Listing 7-7 presents the source code for the convolution neural network and Figure 7-5 illustrates the computation graph. The following points are to be noted:

1. The overall network consists of two convolution-detector blocks, followed by a max pooling layer which in turn is followed by a two layer fully connected network (refer to Chapter 5).
2. The size of the kernel is 3×3.
3. The pooling operation is done over sections of dimensionality 2×2.
4. There are a number of dropout layers, which are basically a form of regularization (refer to Chapter 1) and operate by randomly turning off a certain number of units. The parameter 0.25 indicates the faction of inputs that will be randomly dropped.
5. The flatten layers convert the input of any dimensionality to a dimensionality of $1 \times n$. So, for instance, an input of dimensionality $2 \times 2 \times 3$ gets converted to a directionality of 1×12.
6. The output layer is softmax and the loss function is categorical entropy, as is appropriate for a multiclassification problem (refer to Chapter 3).
7. The model is fit using adadelta (refer to Chapter 8) and, for the purposes of illustration, we set the epochs to 1 (ideally it's set to much more than that).

Listing 7-8. LSTM

```
import numpy as np
from keras.preprocessing import sequence
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM
from keras.datasets import imdb
from keras.utils.visualize_util import plot

max_features = 20000
maxlen = 80
batch_size = 32

# Prepare dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=max_features)
X_train = sequence.pad_sequences(X_train, maxlen=maxlen)
X_test = sequence.pad_sequences(X_test, maxlen=maxlen)

# LSTM
model = Sequential()
model.add(Embedding(max_features, 128, dropout=0.2))
model.add(LSTM(128, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1))
model.add(Activation('sigmoid'))

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

```
# Training
model.fit(X_train, y_train, batch_size=batch_size, verbose=1, nb_epoch=1, validation_
data=(X_test, y_test))

# Evalaution
score = model.evaluate(X_test, y_test, batch_size=batch_size)
print "Test Metrics:", zip(model.metrics_names, score)
plot(model, to_file='s8.png', show_shapes=True)

# Output
# Train on 25000 samples, validate on 25000 samples
# Epoch 1/1
# 25000/25000 [==============================] - 165s - loss: 0.5286 - acc: 0.7347 - val_
loss: 0.4391 - val_acc: 0.8076
# 25000/25000 [==============================] - 33s
# Test Metrics: [('loss', 0.43908300422668456), ('acc', 0.80759999999999998)]
```

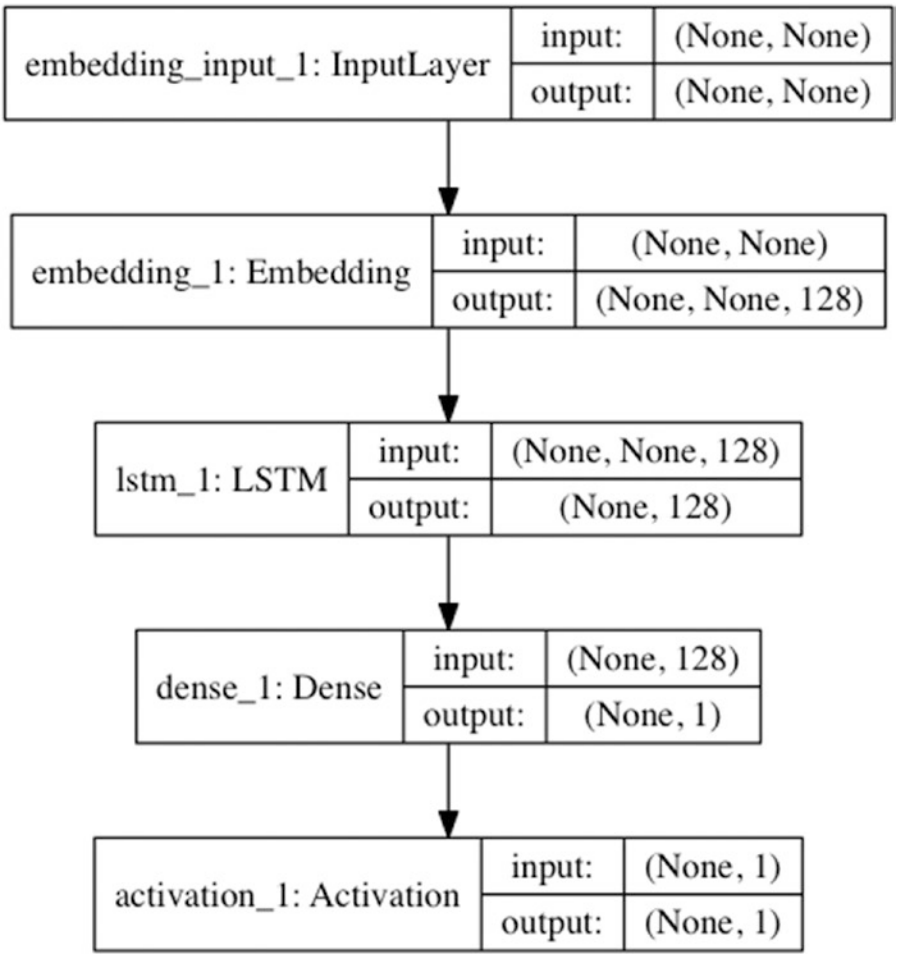

Figure 7-6. *LSTM*

Let us now look at the construct Keras provides to build LSTM Networks, introduced in Chapter 6 (refer to Listing 7-8 and Figure 7-6). The data set we will be using is from IMDB and represents 25,000 reviews from IMDB categorized as positive or negative, making this a binary sequence classification problem. The data is preprocessed to contain only frequent words (the words are actually represented as integers). Listing 7-8 presents the source code and Figure 7-6 illustrates the computational graph. Keras provides a fairly high-level construct for LSTMs, which allows users to construct LSTM models.

Summary

In this chapter we covered the basics of using Keras, using a number of small and simple examples. We encourage the reader to experiment with the examples. Keras has extensive documentation, which is recommended further reading.