CHAPTER 2

Understanding and Working with Keras

Keras is a compact and easy-to-learn high-level Python library for deep learning that can run on top of TensorFlow (or Theano or CNTK). It allows developers to focus on the main concepts of deep learning, such as creating layers for neural networks, while taking care of the nitty-gritty details of tensors, their shapes, and their mathematical details. TensorFlow (or Theano or CNTK) has to be the back end for Keras. You can use Keras for deep learning applications without interacting with the relatively complex TensorFlow (or Theano or CNTK). There are two major kinds of framework: the sequential API and the functional API. The sequential API is based on the idea of a sequence of layers; this is the most common usage of Keras and the easiest part of Keras. The sequential model can be considered as a linear stack of layers.

In short, you create a sequential model where you can easily add layers, and each layer can have convolution, max pooling, activation, dropout, and batch normalization. Let's go through major steps to develop deep learning models in Keras.

N. K. Manaswi, *Deep Learning with Applications Using Python*,
https://doi.org/10.1007/978-1-4842-3516-4_2

Major Steps to Deep Learning Models

The four core parts of deep learning models in Keras are as follows:

1. Define the model. Here you create a sequential model and add layers. Each layer can contain one or more convolution, pooling, batch normalization, and activation function.
2. Compile the model. Here you apply the loss function and optimizer before calling the `compile()` function on the model.
3. Fit the model with training data. Here you train the model on the test data by calling the `fit()` function on the model.
4. Make predictions. Here you use the model to generate predictions on new data by calling functions such as `evaluate()` and `predict()`.

There are eight steps to the deep learning process in Keras:

1. Load the data.
2. Preprocess the data.
3. Define the model.
4. Compile the model.
5. Fit the model.
6. Evaluate the model.
7. Make the predictions.
8. Save the model.

Load Data

Here is how you load data:

```
# Importing modules
import numpy as np
import os
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import adam
from keras.utils import np_utils
```

```
#Load Data
np.random.seed(100) # for reproducibility
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

#cifar-10 has images of airplane, automobile, bird, cat,
# deer, dog, frog, horse, ship and truck ( 10 unique labels)
# For each image. width = 32, height =32, Number of channels(RGB) = 3
```

Preprocess the Data

Here is how you preprocess data:

```
#Preprocess the data
#Flatten the data, MLP doesn't use the 2D structure of the data. 3072 = 3*32*32
X_train = X_train.reshape(50000, 3072) # 50,000 images for training
X_test = X_test.reshape(10000, 3072) # 10,000 images for test

# Gaussian Normalization( Z- score)
X_train = (X_train- np.mean(X_train))/np.std(X_train)
X_test = (X_test- np.mean(X_test))/np.std(X_test)
```

```
# Convert class vectors to binary class matrices (ie one-hot vectors)
labels = 10 #10 unique labels(0-9)
Y_train = np_utils.to_categorical(y_train, labels)
Y_test = np_utils.to_categorical(y_test, labels)
```

Define the Model

Sequential models in Keras are defined as a sequence of layers. You create a sequential model and then add layers. You need to ensure the input layer has the right number of inputs. Assume that you have 3,072 input variables; then you need to create the first hidden layer with 512 nodes/neurons. In the second hidden layer, you have 120 nodes/neurons. Finally, you have ten nodes in the output layer. For example, an image maps onto ten nodes that shows the probability of being label1 (airplane), label2 (automobile), label3 (cat), …, label10 (truck). The node of highest probability is the predicted class/label.

```
#Define the model achitecture
model = Sequential()
model.add(Dense(512, input_shape=(3072,))) # 3*32*32 = 3072
model.add(Activation('relu'))
model.add(Dropout(0.4)) # Regularization
model.add(Dense(120))
model.add(Activation('relu'))
model.add(Dropout(0.2))# Regularization
model.add(Dense(labels)) #Last layer with 10 outputs, each output per class
model.add(Activation('sigmoid'))
```

One image has three channels (RGB), and in each channel, the image has 32×32 = 1024 pixels. So, each image has 3×1024 = 3072 pixels (features/X/inputs).

With the help of 3,072 features, you need to predict the probability of label1 (Digit 0), label2 (Digit 1), and so on. This means the model predicts ten outputs (Digits 0–9) where each output represents the probability of the corresponding label. The last activation function (sigmoid, as shown earlier) gives 0 for nine outputs and 1 for only one output. That label is the predicted class for the image (Figure 2-1).

For example, 3,072 features ➤ 512 nodes ➤ 120 nodes ➤ 10 nodes.

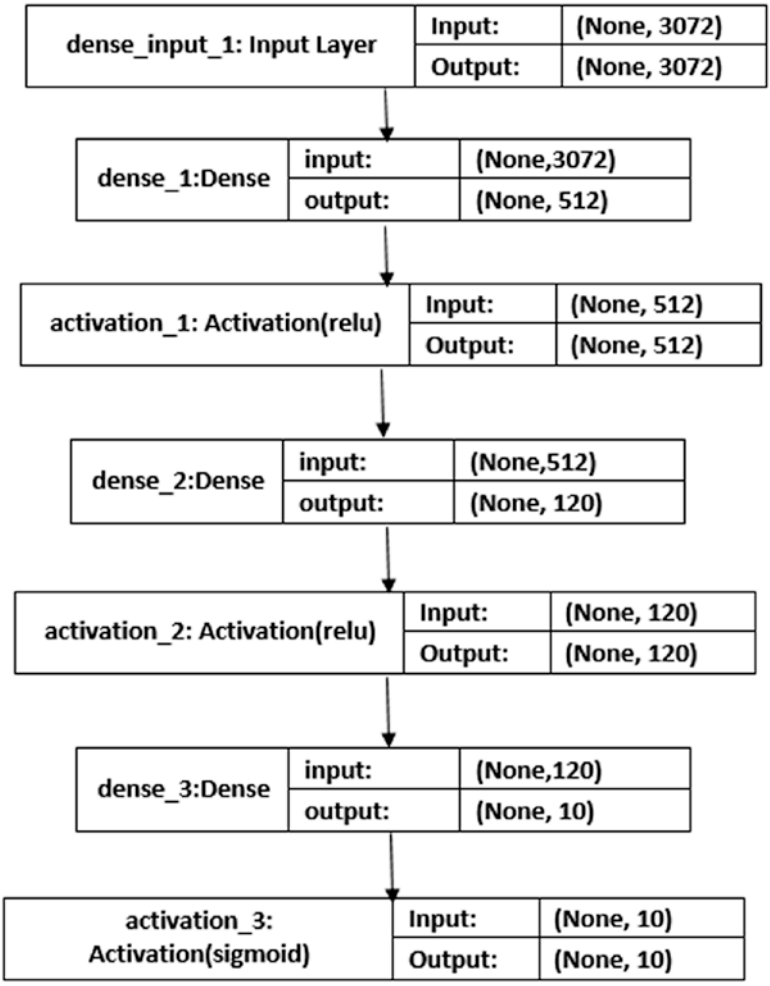

Figure 2-1. *Defining the model*

The next question is, how do you know the number of layers to use and their types? No one has the exact answer. What's best for evaluation metrics is that you decide the optimum number of layers and the parameters and steps in each layer. A heuristics approach is also used. The best network structure is found through a process of trial-and-error experimentation. Generally, you need a network large enough to capture the structure of the problem.

In this example, you will use a fully connected network structure with three layers. A dense class defines fully connected layers.

In this case, you initialize the network weights to a small random number generated from a uniform distribution (uniform) in this case between 0 and 0.05 because that is the default uniform weight initialization in Keras. Another traditional alternative would be normal for small random numbers generated from a Gaussian distribution. You use or snap to a hard classification of either class with a default threshold of 0.5. You can piece it all together by adding each layer.

Compile the Model

Having defined the model in terms of layers, you need to declare the loss function, the optimizer, and the evaluation metrics. When the model is proposed, the initial weight and bias values are assumed to be 0 or 1, a random normally distributed number, or any other convenient numbers. But the initial values are not the best values for the model. This means the initial values of weight and bias are not able to explain the target/label in terms of predictors (Xs). So, you want to get the optimal value for the model. The journey from initial values to optimal values needs a motivation, which will minimize the cost function/loss function. The journey needs a path (change in each iteration), which is suggested by the optimizer. The journey also needs an evaluation measurement, or evaluation metrics.

```
# Compile the model
# Use adam as an optimizer
adam = adam(0.01)
# the cross entropy between the true label and the output(softmax) of the model
model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=["accuracy"])
```

Popular loss functions are binary cross entropy, categorical cross entropy, mean_squared_logarithmic_error and hinge loss. Popular optimizers are stochastic gradient descent (SGD), RMSProp, adam, adagrad, and adadelta. Popular evaluation metrics are accuracy, recall, and F1 score.

In short, this step is aimed at tuning the weights and biases based on loss functions through iterations based on the optimizer evaluated by metrics such as accuracy.

Fit the Model

Having defined and compiled the model, you need to make predications by executing the model on some data. Here you need to specify the epochs; these are the number of iterations for the training process to run through the data set and the batch size, which is the number of instances that are evaluated before a weight update. For this problem, the program will run for a small number of epochs (10), and in each epoch, it will complete 50(=50,000/1,000) iterations where the batch size is 1,000 and the training data set has 50,000 instances/images. Again, there is no hard rule to select the batch size. But it should not be very small, and it should be much less than the size of the training data set to consume less memory.

```
#Make the model learn ( Fit the model)
model.fit(X_train, Y_train,batch_size=1000, nb_epoch=10,validation_data=(X_test, Y_test))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
 1000/50000 [..............................] - ETA: 6s - loss: 2.3028 - acc: 0.1060
C:\ProgramData\Anaconda3\lib\site-packages\keras\models.py:848: UserWarning: The `nb_epoch` argument in `fit` has been renamed
`epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '
50000/50000 [==============================] - 6s - loss: 2.3030 - acc: 0.0974 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 2/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.1012 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 3/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0972 - val_loss: 2.3026 - val_acc: 0.1000
Epoch 4/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0997 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 5/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0975 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 6/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0986 - val_loss: 2.3028 - val_acc: 0.1000
Epoch 7/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0995 - val_loss: 2.3028 - val_acc: 0.1000
Epoch 8/10
50000/50000 [==============================] - 7s - loss: 2.3028 - acc: 0.0983 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 9/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0998 - val_loss: 2.3027 - val_acc: 0.1000
Epoch 10/10
50000/50000 [==============================] - 7s - loss: 2.3029 - acc: 0.0972 - val_loss: 2.3027 - val_acc: 0.1000
<keras.callbacks.History at 0x2870136eef0>
```

Evaluate Model

Having trained the neural networks on the training data sets, you need to evaluate the performance of the network. Note that this will only give you an idea of how well you have modeled the data set (e.g., the train accuracy), but you won't know how well the algorithm might perform on new data. This is for simplicity, but ideally, you could separate your data into train and test data sets for the training and evaluation of your model. You can evaluate your model on your training data set using the `evaluation()` function on your model and pass it the same input and output used to train the model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

```
#Evaluate how the model does on the test set
score = model.evaluate(X_test, Y_test, verbose=0)
#Accuracy Score
print('Test accuracy:', score[1])
```

Prediction

Once you have built and evaluated the model, you need to predict for unknown data.

```
#Predict digit(0-9) for test Data
model.predict_classes(X_test)
```

```
 9888/10000 [============================>.] - ETA: 0s
array([3, 8, 8, ..., 3, 4, 7], dtype=int64)
```

Save and Reload the Model

Here is the final step:

```
#Saving the model
model.save('model.h5')
jsonModel = model.to_json()
model.save_weights('modelWeight.h5')
```

```
#Load weight of the saved model
modelWt = model.load_weights('modelWeight.h5')
```

Optional: Summarize the Model

Now let's see how to summarize the model.

```
#Summary of the model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 512)               1573376
_________________________________________________________________
activation_7 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_5 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_8 (Dense)              (None, 120)               61560
_________________________________________________________________
activation_8 (Activation)    (None, 120)               0
_________________________________________________________________
dropout_6 (Dropout)          (None, 120)               0
_________________________________________________________________
dense_9 (Dense)              (None, 10)                1210
_________________________________________________________________
activation_9 (Activation)    (None, 10)                0
=================================================================
Total params: 1,636,146
Trainable params: 1,636,146
Non-trainable params: 0
_________________________________________________________________
```

Additional Steps to Improve Keras Models

Here are some more steps to improve your models:

1. Sometimes, the model building process does not complete because of a vanishing or exploding gradient. If this is the case, you should do the following:

```
from keras.callbacks import EarlyStopping
early_stopping_ monitor = EarlyStopping(patience=2)
model.fit(x_train, y_train, batch_size=1000, epochs=10,
validation_data=(x_test, y_test),
callbacks=[early_stopping_monitor])
```

2. Model the output shape.

```
#Shape of the n-dim array (output of the model
at the current position)
  model.output_shape
```

3. Model the summary representation.

```
model.summary()
```

4. Model the configuration.

```
model.get_config()
```

5. List all the weight tensors in the model.

```
model.get_weights()
```

Here I am sharing the complete code for the Keras model. Can you attempt to explain it?

```
# A TYPING DEEP LEARNING MODEL WITH KERAS
      import numpy as np
      from keras.models import Sequential
      from keras.layers import Dense

# Loading Data

     data = np.random.random((500,100))
     labels = np.random.randint(2,size=(500,1))


# Create model
          model = Sequential()
          model.add(Dense(12, input_dim=8, activation='relu'))
          model.add(Dense(8, activation='relu'))
         model.add(Dense(1, activation='sigmoid'))

# Compile model

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

# Fit the model

     model.fit(X[train], Y[train], epochs=150, batch_size=10, verbose=0)

# Evaluate the model

     scores = model.evaluate(X[test], Y[test], verbose=0)
    print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
   cvscores.append(scores[1] * 100) print("%.2f%% (+/- %.2f%%)" %
(numpy.mean(cvscores), numpy.std(cvscores)))

# Predict
      predictions = model.predict(data)
```

Keras with TensorFlow

Keras provides high-level neural networks by leveraging a powerful and lucid deep learning library on top of TensorFlow/Theano. Keras is a great addition to TensorFlow as its layers and models are compatible with pure-TensorFlow tensors. Moreover, it can be used alongside other TensorFlow libraries.

Here are the steps involved in using Keras for TensorFlow:

1. Start by creating a TensorFlow session and registering it with Keras. This means Keras will use the session you registered to initialize all the variables that it creates internally.

```
import TensorFlow as tf
sess = tf.Session()
from keras import backend as K
K.set_session(sess)
```

2. Keras modules such as the model, layers, and activation are used to build models. The Keras engine automatically converts these modules into the TensorFlow-equivalent script.
3. Other than TensorFlow, Theano and CNTK can be used as back ends to Keras.
4. A TensorFlow back end has the convention of making the input shape (to the first layer of your network) in depth, height, width order, where depth can mean the number of channels.

5. You need to configure the `keras.json` file correctly so that it uses the TensorFlow back end. It should look something like this:

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

In next chapters, you will learn how to leverage Keras for working on CNN, RNN, LSTM, and other deep learning activities.