CHAPTER 5

Regression to MLP in Keras

You have been working on regression while solving machine learning applications. Linear regression and nonlinear regression are used to predict numeric targets, while logistic regression and other classifiers are used to predict non-numeric target variables. In this chapter, I will discuss the evolution of multilayer perceptrons.

Specifically, you will compare the accuracy generated by different models with and without using Keras.

Log-Linear Model

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
####### Log-Linear Model ########################################
from sklearn.datasets import load_iris
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LogisticRegressionCV
from sklearn.linear_model import LinearRegression
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Activation
```


N. K. Manaswi, *Deep Learning with Applications Using Python*,
https://doi.org/10.1007/978-1-4842-3516-4_5

You will be using the Iris data set as the source of data. So, load the data set from Seaborn.

```
# Load the iris dataset from seaborn.
iris = load_iris()
```

The Iris data set has five attributes. You will be using the first four attributes to predict the species, whose class is defined in the fifth attribute of the data set.

```
# Use the first 4 variables to predict the species.
X, y = iris.data[:, :4], iris.target
```

Using `scikit-learn`'s function, split the testing and training data sets.

```
# Split both independent and dependent variables in half
# for cross-validation
train_X, test_X, train_y, test_y = train_test_split(X, y, train_size=0.5, random_state=0)
#print(type(train_X),len(train_y),len(test_X),len(test_y))
```

```
###################################
# scikit Learn for (Log) Linear Regression #
###################################
```

Use the `model.fit` function to train the model with the training data set.

```
# Train a scikit-learn log-regression model
# lr =LogisticRegressionCV
# Train a scikit-learn linear-regression model
lr = LinearRegression()
lr.fit(train_X, train_y)
```

As the model is trained, you can predict the output of the test set.

```
# Test the model. Print the accuracy on the test data
pred_y = lr.predict(test_X)
#print("Accuracy is {:.2f}".format(lr.score(test_X, test_y)))
```

Keras Neural Network for Linear Regression

Now, let's build a Keras neural network model for linear regression.

```
# Build the keras model
model = Sequential()
# 4 features in the input layer (the four flower measurements)
# 16 hidden units
model.add(Dense(16, input_shape=(4,)))
model.add(Activation('sigmoid'))
# 3 classes in the ouput layer (corresponding to the 3 species)
model.add(Dense(3))
model.add(Activation('softmax'))
```

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

Use the `model.fit` function to train the model with the training data set.

```
# Fit/Train the keras model
model.fit(train_X, train_y, verbose=1, batch_size=1, nb_epoch=100)
```

As the model is trained, you can predict the output of the test set.

```
# Test the model. Print the accuracy on the test data
loss, accuracy = model.evaluate(test_X, test_y, verbose=0)
print("\nAccuracy is using keras prediction  {:.2f}".format(accuracy))
```

Print the accuracy obtained by both models.

```
print("\nAccuracy is using keras prediction  {:.2f}".format(accuracy))

print("Accuracy is using regression  {:.2f}".format(lr.score(test_X, test_y)))
```

If you run the code, you will see the following output:

```
Using TensorFlow backend.

Epoch 1/100

75/75 [==============================] - 0s - loss: 1.2947 - acc: 0.4533

Epoch 2/100

75/75 [==============================] - 0s - loss: 1.0353 - acc: 0.6400

Epoch 3/100

75/75 [==============================] - 0s - loss: 0.8930 - acc: 0.6533

...                    ...                    ...

...                    ...                    ...

...                    ...                    ...

...                    ...                    ...

Epoch 99/100

75/75 [==============================] - 0s - loss: 0.1186 - acc: 0.9733

Epoch 100/100

75/75 [==============================] - 0s - loss: 0.1167 - acc: 0.9867


Accuracy is using keras prediction  0.99

Accuracy is using regression  0.89
```

Logistic Regression

In this section, I will share an example for the logistic regression so you can compare the code in `scikit-learn` with that in Keras (see Figure 5-1).

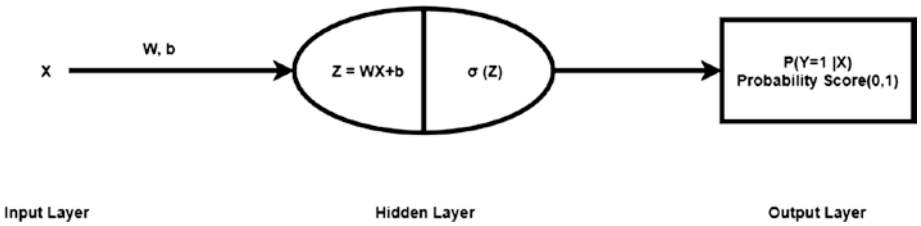

Figure 5-1. *Logistic regression used for classification*

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
from sklearn.datasets import load_iris
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LogisticRegressionCV
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.utils import np_utils
```

You will be using the Iris data set as the source of data. So, load the data set from scikit-learn.

```
# Load Data and Prepare data
iris = load_iris()
X, y = iris.data[:, :4], iris.target
```

Using `scikit-learn`'s function, split the testing and training data sets.

```
# Load Data and Prepare data
iris = load_iris()
X, y = iris.data[:, :4], iris.target
```

scikit-learn for Logistic Regression

Use the `model.fit` function to train the model with the training data set. After the model is trained, you can predict the output of the test set.

```
##############################
# scikit Learn for  Logistic Regression
##############################
lr = LogisticRegressionCV()
lr.fit(train_X, train_y)
pred_y = lr.predict(test_X)
print("Test fraction correct (LR-Accuracy) = {:.2f}".format(lr.score(test_X, test_y)))
```

##

Keras Neural Network for Logistic Regression

One-hot encoding transforms features to a format that works better with the classification and regression algorithms.

```
################################
# Keras Neural Network for Logistic Regression
################################

# Make ONE-HOT enconding for converting into categorical variable
def one_hot_encode_object_array(arr):
    uniques, ids = np.unique(arr, return_inverse=True)
    return np_utils.to_categorical(ids, len(uniques))
```

```
# Dividing data into train and test data
train_y_ohe = one_hot_encode_object_array(train_y)
test_y_ohe = one_hot_encode_object_array(test_y)
#Creating a model
model = Sequential()
model.add(Dense(16, input_shape=(4,)))
model.add(Activation('sigmoid'))
model.add(Dense(3))
model.add(Activation('softmax'))
```

```
# Compiling the model
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')
```

Use the `model.fit` function to train the model with the training data set.

```
# Actual modelling
model.fit(train_X, train_y_ohe, verbose=0, batch_size=1, nb_epoch=100)
```

Use the `model.evaluate` function to evaluate the performance of the model.

```
score, accuracy = model.evaluate(test_X, test_y_ohe, batch_size=16, verbose=0)
```

Print the accuracy obtained by both models.

Accuracy for scikit-learn based model

```
print("\n Test fraction correct (LR-Accuracy) logistic regression = {:.2f}".format(lr.score(test_X, test_y)))
```

The accuracy is 0.83.

Accuracy for keras model

```
print("Test fraction correct (NN-Accuracy) keras  = {:.2f}".format(accuracy))
```

The accuracy is 0.99.

If you run the code, you will see the following output:

```
Using TensorFlow backend.
Test fraction correct (LR-Accuracy) logistic regression =
0.83
Test fraction correct (NN-Accuracy) keras  = 0.99
Epoch 1/100
75/75 [==============================] - 0s - loss: 1.2947 -
acc: 0.4533
Epoch 2/100
75/75 [==============================] - 0s - loss: 1.0353 -
acc: 0.6400
Epoch 3/100
75/75 [==============================] - 0s - loss: 0.8930 -
acc: 0.6533
...                                   ...                   ...
...                                   ...                   ...
...                                   ...                   ...
...                                   ...                   ...
Epoch 99/100
75/75 [==============================] - 0s - loss: 0.1186 -
acc: 0.9733
Epoch 100/100
75/75 [==============================] - 0s - loss: 0.1167 -
acc: 0.9867

Accuracy is using keras prediction  0.99
Accuracy is using regression  0.89
```

To give the real-life example, I will discuss some code that uses the Fashion MNIST data set, which is a data set of Zalando.com's images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28×28 grayscale image associated with a label from ten classes.

Fashion MNIST Data: Logistic Regression in Keras

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
from __future__ import print_function
from keras.models import load_model
import keras
import fashion_mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
import numpy as np

batch_size = 128
num_classes = 10
epochs = 2
```

As mentioned, you will be using the Fashion MNIST data set. Store the data and the label in two different variables.

```
# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

Normalize the data set, as shown here:

```
#Gaussian Normalization of the dataset
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
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Define the model, as shown here:

```
#Building a model architecture
model = Sequential()
model.add(Dense(256, activation='elu', input_shape=(784,)))
model.add(Dropout(0.4))
model.add(Dense(512, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.summary()

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

model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test))
```

Save the model in an `.h5` file (so that you can use it later directly with the `load_model()` function from keras.models) and print the accuracy of the model in the test set, as shown here:

```
#saving the model using the 'model.save' function
model.save('my_model.h5')
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

If you run the previous code, you will see the following output:

```
('train-images-idx3-ubyte.gz', <http.client.HTTPMessage object
at 0x00000171338E2B38>)

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_59 (Dense)             (None, 256)               200960
_________________________________________________________________
dropout_10 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_60 (Dense)             (None, 512)               131584
_________________________________________________________________
dense_61 (Dense)             (None, 10)                5130
=================================================================
Total params: 337,674
Trainable params: 337,674
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/2
60000/60000 [==============================] - loss: 0.5188 -
acc: 0.8127 - val_loss: 0.4133 - val_acc: 0.8454
Epoch 2/2
60000/60000 [==============================] - loss: 0.3976 -
acc: 0.8545 - val_loss: 0.4010 - val_acc: 0.8513
Test loss: 0.400989927697
Test accuracy: 0.8513
```

MLPs on the Iris Data

A multilayer perceptron is a minimal neural network model. In this section, I'll show you the code.

Write the Code

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
##########MLP on iris data ####################

import pandas as pd
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.utils import np_utils
```

Load the data set by reading a CSV file using Pandas.

```
#Load and Prepare Data
datatrain = pd.read_csv('./Datasets/iris/iris_train.csv')
```

Assign numeric values to the classes of the data set.

```
#change string value to numeric
datatrain.set_value(datatrain['species']=='Iris-setosa',['species'],0)
datatrain.set_value(datatrain['species']=='Iris-versicolor',['species'],1)
datatrain.set_value(datatrain['species']=='Iris-virginica',['species'],2)
datatrain = datatrain.apply(pd.to_numeric)
```

Convert the data frame to an array.

```
#change dataframe to array
datatrain_array = datatrain.as_matrix()
```

Split the data and the target and store them in two different variables.

```
# split x and y (feature and target)
xtrain = datatrain_array[:,:4]
ytrain = datatrain_array[:,4]
```

Change the target format using Numpy.

```
#change target format
ytrain = np_utils.to_categorical(ytrain)
```

Build a Sequential Keras Model

Here you will build a multilayer perceptron model with one hidden layer.

- *Input layer*: The input layer contains four neurons, representing the features of an iris (sepal length, etc.).
- *Hidden layer*: The hidden layer contains ten neurons, and the activation uses ReLU.
- *Output layer*: The output layer contains three neurons, representing the classes of the Iris softmax layer.

```
#Build Keras model
#Multilayer perceptron model, with one hidden layer.
#Input layer : 4 neuron, represents the feature of Iris(Sepal Length etc)
#Hidden layer : 10 neuron, activation using ReLU
#Output layer : 3 neuron, represents the class of Iris, Softmax Layer
model = Sequential()
model.add(Dense(output_dim=10, input_dim=4))
model.add(Activation("relu"))
model.add(Dense(output_dim=3))
model.add(Activation("softmax"))
```

Compile the model and choose an optimizer and loss function for training and optimizing your data, as shown here:

```
#Compile model :choose optimizer and loss function
#optimizer = stochastic gradient descent with no batch-size
#loss function = categorical cross entropy
#learning rate = default from keras.optimizer.SGD, 0.01
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

Train the model using the `model.fit` function, as shown here:

```
#train
model.fit(xtrain, ytrain, nb_epoch=100, batch_size=120)
```

Load and prepare the test data, as shown here:

```
## Evaluate on test data
#Load and Prepare Data
datatest = pd.read_csv('./Datasets/iris/iris_test.csv')
```

Convert the string value to a numeric value, as shown here:

```
#change string value to numeric
datatest.set_value(datatest['species']=='Iris-setosa',['species'],0)
datatest.set_value(datatest['species']=='Iris-versicolor',['species'],1)
datatest.set_value(datatest['species']=='Iris-virginica',['species'],2)
datatest = datatest.apply(pd.to_numeric)
```

Convert the data frame to an array, as shown here:

```
#change dataframe to array
datatest_array = datatest.as_matrix()
```

Split x and y, in other words, the feature set and target set, as shown here:

```
#split x and y (feature and target)
xtest= datatest_array[:,:4]
ytest = datatest_array[:,4]
```

Make a prediction on the trained model, as shown here:

```
#get prediction
classes = model.predict_classes(xtest, batch_size=120)
```

Calculate the accuracy, as shown here:

```
#get accuration
accuration = np.sum(classes == ytest)/30.0 * 100
```

Print the accuracy generated by the model, as shown here:

```
print("Test Accuration : " + str(accuration) + '%')
print("Prediction :")
print(classes)
print("Target :")
print(np.asarray(ytest,dtype="int32"))
```

If you run the code, you will see the following output:

```
Epoch 1/100
120/120 [==============================] - 0s - loss: 2.7240 -
acc: 0.3667
Epoch 2/100
120/120 [==============================] - 0s - loss: 2.4166 -
acc: 0.3667
Epoch 3/100
120/120 [==============================] - 0s - loss: 2.1622 -
acc: 0.4083
Epoch 4/100
120/120 [==============================] - 0s - loss: 1.9456 -
acc: 0.6583
```

```
Epoch 98/100
120/120 [==============================] - 0s - loss: 0.5571 -
acc: 0.9250
Epoch 99/100
120/120 [==============================] - 0s - loss: 0.5554 -
acc: 0.9250
Epoch 100/100
120/120 [==============================] - 0s - loss: 0.5537 -
acc: 0.9250
```

MLPs on MNIST Data (Digit Classification)

MNIST is the standard data set to predict handwritten digits. In this section, you will see how you can apply the concept of multilayer perceptrons and make a handwritten digit recognition system.

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
##########MLP : MNIST Data (Digit Classification) ########################

import numpy as np
import os
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop
from keras.utils import np_utils
```

Sone important variables are defined.

```
np.random.seed(100) # for reproducibility
batch_size = 128 #Number of images used in each optimization step
nb_classes = 10 #One class per digit
nb_epoch = 20 #Number of times the whole data is used to learn
```

Load the data set using the `mnist.load_data()` function.

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

#Flatten the data, MLP doesn't use the 2D structure of the data. 784 = 28*28
X_train = X_train.reshape(60000, 784) # 60,000 digit images
X_test = X_test.reshape(10000, 784)
```

The types of the training set and the test set are converted to `float32`.

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

The data sets are normalized; in other words, they are set to a Z-score.

```
# 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)
```

Display the number of the training samples present in the data set and also the number of test sets available.

```
#Display number of training and test instances
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

Convert class vectors to binary class matrices.

```
# convert class vectors to binary class matrices (ie one-hot vectors)
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

Define the sequential model of the multilayer perceptron.

```
#Define the model achitecture
model = Sequential()
model.add(Dense(512, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(0.2)) # Regularization
model.add(Dense(120))
model.add(Activation('relu'))
model.add(Dropout(0.2))
model.add(Dense(10)) #Last layer with one output per class
model.add(Activation('softmax')) #We want a score simlar to a probability for each class
```

Use an optimizer.

```
#Use rmsprop as an optimizer
rms = RMSprop()
```

The function to optimize is the cross entropy between the true label and the output (softmax) of the model.

```
#The function to optimize is the cross entropy between the true label and the output (softmax) of the model
model.compile(loss='categorical_crossentropy', optimizer=rms, metrics=["accuracy"])
```

Use the `model.fit` function to train the model.

```
#Make the model learn
model.fit(X_train, Y_train,
batch_size=batch_size, nb_epoch=nb_epoch,
verbose=2,
validation_data=(X_test, Y_test))
```

Using the model, evaluate the function to evaluate the performance of the model.

```
#Evaluate how the model does on the test set
score = model.evaluate(X_test, Y_test, verbose=0)
```

Print the accuracy generated in the model.

```
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

If you run the code, you will get the following output:
60000 train samples

```
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
13s - loss: 0.2849 - acc: 0.9132 - val_loss: 0.1149 - val_acc:
0.9652
Epoch 2/20
11s - loss: 0.1299 - acc: 0.9611 - val_loss: 0.0880 - val_acc:
0.9741
Epoch 3/20
11s - loss: 0.0998 - acc: 0.9712 - val_loss: 0.1121 - val_acc:
0.9671
Epoch 4/20
Epoch 18/20
14s - loss: 0.0538 - acc: 0.9886 - val_loss: 0.1241 - val_acc:
0.9814
Epoch 19/20
12s - loss: 0.0522 - acc: 0.9888 - val_loss: 0.1154 - val_acc:
0.9829
Epoch 20/20
13s - loss: 0.0521 - acc: 0.9891 - val_loss: 0.1183 - val_acc:
0.9824
Test score: 0.118255248802
Test accuracy: 0.9824
```

Now, it is time to create a data set and use a multilayer perceptron. Here you will create your own data set using the random function and run the multilayer perceptron model on the generated data.

MLPs on Randomly Generated Data

Create a new Python file and import the following packages. Make sure you have Keras installed on your system.

```
#####MLP on randomly generated Data ##############
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD
import numpy as np
```

Generate the data using the `random` function.

```
# Generate dummy data
x_train = np.random.random((1000, 20))
# Y having 10 possible categories
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)
```

Create a sequential model.

```
#Create a model
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, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

Compile the model.

```
#Compile the model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',optimizer=sgd,metrics=['accuracy'])
```

Use the `model.fit` function to train the model.

```
# Fit the model
model.fit(x_train, y_train,epochs=20,batch_size=128)
```

Evaluate the performance of the model using the `model.evaluate` function.

```
# Evaluate the model
score = model.evaluate(x_test, y_test, batch_size=128)
```

If you run the code, you will get the following output:

```
Epoch 1/20
1000/1000 [==============================] - 0s - loss:
2.4432 - acc: 0.0970
Epoch 2/20
1000/1000 [==============================] - 0s - loss:
2.3927 - acc: 0.0850
Epoch 3/20
1000/1000 [==============================] - 0s - loss:
2.3361 - acc: 0.1190
Epoch 4/20
1000/1000 [==============================] - 0s - loss:
2.3354 - acc: 0.1000
Epoch 19/20
1000/1000 [==============================] - 0s - loss:
2.3034 - acc: 0.1160
Epoch 20/20
1000/1000 [==============================] - 0s - loss:
2.3055 - acc: 0.0980
100/100 [==============================] - 0s
```

In this chapter, I discussed how to build linear, logistic, and MLP models in Keras in a systemic way.