Model training with Keras

October 14, 2017

1 Modeling a test network using Keras

1.1 Introduction

I will be using keras with dense layers and a Sequential model. Herein, we will be using standard data from this archive for *diabetic data*. First, let us import the necessary data.

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense
        import numpy as np
        import matplotlib.pyplot as plt
        np.random.seed(5)
```

```
Using TensorFlow backend.
```

1.2 Importing data and parsing

Now that we have import all the required files, let us first import the dataset

```
In [2]: import os
        filename = 'dataset.csv'
        assert(filename) in os.listdir()

        raw_data = open(filename).read().strip(' ').strip('\n').split('\n')
        dataset = [[float(x) for x in y.split(',')] for y in raw_data ]
        dataset = np.array(dataset)
        dataset.shape
```

```
Out[2]: (768, 9)
```

After importing the dataset, we can move to making the X and y features and then assembling the model.

```
In [3]: X = dataset[:, :-1]
        y = dataset[:, -1]
        y = y.reshape(y.shape[0], 1)
        'X = ', X
```

```
Out[3]: ('X = ',
         array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
                [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
                [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
                ...,
                [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
                [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
                [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]]))
```

1.3 Assembling the model

After the data has been imported, we can start assembling the model. According to our input features, we will model the network as a *3-layer*, network with the **reLu** activation function. Our first layer has **8 input features** (matching the shape of the X vector); it has a *relu* activation function and consists of *12 nodes*. The second layer has *12 nodes* and a *relu* activation function. Being a binary classification problem, we can model our output layer to have either a *sigmoid* activation function or a *tanh* function.

```
In [4]: print(X.shape, y.shape)
        # Our model is as follows
        network = Sequential()
        # adding the first layer
        network.add(Dense(12, input_dim=X.shape[-1], activation='relu'))
        # adding the second layer
        network.add(Dense(12, activation='relu'))
        # adding the output layer
        network.add(Dense(1, activation='sigmoid'))
```

```
(768, 8) (768, 1)
```

Now that our network has been setup, we can feed it data to train. We can divide our dataset into 2 parts, the training data and the test data. Even though *keras* does this on its own, it is still a good practice when you have plentiful data.

1.4 Training the model

We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training.

In this case, we will use logarithmic loss, which for a binary classification problem is defined in Keras as "binary_crossentropy". We will also use the efficient gradient descent algorithm "adam" for no other reason that it is an efficient default. Learn more about the Adam optimization algorithm in the paper "Adam: A Method for Stochastic Optimization".

```
In [5]: training_data = X[:int(0.6 * X.shape[0])]
        training_y = y[:int(0.6 * y.shape[0])]
        network.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
        iterations = int(150)
```

```
        # the batch size is used for the mini-batch grad descent which almost always
        # fastens the progress
        batch_size =  int(0.2 * X.shape[0])
```

```
In [6]: network.fit(training_data, training_y, verbose=0, epochs=iterations, batch_size=batch_s
```

```
Out[6]: <keras.callbacks.History at 0x7fea19c99710>
```

1.5 Checking accuracy & Improvement

Now that we have trained the model, we can check it's accuracy on the training set, test set, as well as the entire dataset. A good accuracy is around 80%+, but this can vary greatly according to your use case.

```
In [7]: # adding the scores
        import json
        scores = {
            'test_data' : network.evaluate(X[-int(0.4 * X.shape[0]):], y[-int(0.4 * y.shape[0]]
            'training_data' : network.evaluate(training_data, training_y, verbose=0),
            'dataset' : network.evaluate(X, y, verbose=0)
        }
        print('loss', network.metrics_names[0], '\nAccuracy : ', network.metrics_names[1])
        print([(x, scores[x][1] * 100) for x in scores])
```

```
loss loss
Accuracy :  acc
[('test_data', 66.449511478312246), ('training_data', 70.434782556865528), ('dataset', 68.8802
```

As seen, our accuracy isn't that great. One thing we can do is normalize the data and perhaps use the *tanh* activation function instead of the *sigmoid* function.

```
In [9]: training_data = (training_data - np.mean(training_data, 0)) / np.std(training_data, 0)
        network.fit(training_data, training_y, epochs=iterations, batch_size=batch_size, verbos
```

```
Out[9]: <keras.callbacks.History at 0x7fea184cdf60>
```

```
In [12]: X = (X - np.mean(X, 0)) / np.std(X, 0)
         exec(In[7])
         print("Final network\n", network.summary())
```

```
loss loss
Accuracy :  acc
[('test_data', 76.221498313089924), ('training_data', 74.130434730778575), ('dataset', 74.8697
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12)                108
_________________________________________________________________
```

```
dense_2 (Dense)              (None, 12)               156
_________________________________________________________________
dense_3 (Dense)              (None, 1)                13
=================================================================
Total params: 277
Trainable params: 277
Non-trainable params: 0
_________________________________________________________________
Final network
 None
```

Just by normalizing the data, you see we get a huge improvement. On the whole, our accuracy has jumped from *64%* to *78%*, a whopping **increase of 12%**. We can further improve this accuracy by: - adding more nodes to a layer - adding more layers - using *tanh* activations, etc.

Hyperparameters can be changed according to your use case and different settings may work differently for different people.