DL0101EN-3-1-Regression-with-Keras-py-v1.0

January 15, 2020

Regression Models with Keras

0.1 Introduction

As we discussed in the videos, despite the popularity of more powerful libraries such as PyToch and TensorFlow, they are not easy to use and have a steep learning curve. So, for people who are just starting to learn deep learning, there is no better library to use other than the Keras library.

Keras is a high-level API for building deep learning models. It has gained favor for its ease of use and syntactic simplicity facilitating fast development. As you will see in this lab and the other labs in this course, building a very complex deep learning network can be achieved with Keras with only few lines of code. You will appreciate Keras even more, once you learn how to build deep models using PyTorch and TensorFlow in the other courses.

So, in this lab, you will learn how to use the Keras library to build a regression model.

0.2 Table of Contents

0.3 Download and Clean Dataset

Let's start by importing the pandas and the Numpy libraries.

[1]:
```
import pandas as pd
import numpy as np
```

We will be playing around with the same dataset that we used in the videos.

The dataset is about the compressive strength of different samples of concrete based on the volumes of the different ingredients that were used to make them. Ingredients include:

1. Cement

2. Blast Furnace Slag

3. Fly Ash
4. Water
5. Superplasticizer
6. Coarse Aggregate
7. Fine Aggregate

Let's download the data and read it into a pandas dataframe.

[2]:
```
concrete_data = pd.read_csv('https://s3-api.us-geo.objectstorage.softlayer.net/
↪cf-courses-data/CognitiveClass/DL0101EN/labs/data/concrete_data.csv')
concrete_data.head()
```

[2]:
```
   Cement  Blast Furnace Slag  Fly Ash  Water  Superplasticizer  \
0   540.0                 0.0      0.0  162.0               2.5
1   540.0                 0.0      0.0  162.0               2.5
2   332.5               142.5      0.0  228.0               0.0
3   332.5               142.5      0.0  228.0               0.0
4   198.6               132.4      0.0  192.0               0.0

   Coarse Aggregate  Fine Aggregate  Age  Strength
0            1040.0           676.0   28     79.99
1            1055.0           676.0   28     61.89
2             932.0           594.0  270     40.27
3             932.0           594.0  365     41.05
4             978.4           825.5  360     44.30
```

So the first concrete sample has 540 cubic meter of cement, 0 cubic meter of blast furnace slag, 0 cubic meter of fly ash, 162 cubic meter of water, 2.5 cubic meter of superplaticizer, 1040 cubic meter of coarse aggregate, 676 cubic meter of fine aggregate. Such a concrete mix which is 28 days old, has a compressive strength of 79.99 MPa.

Let's check how many data points we have.

[3]:
```
concrete_data.shape
```

[3]:
```
(1030, 9)
```

So, there are approximately 1000 samples to train our model on. Because of the few samples, we have to be careful not to overfit the training data.

Let's check the dataset for any missing values.

[4]:
```
concrete_data.describe()
```

[4]:
```
            Cement  Blast Furnace Slag      Fly Ash        Water  \
count  1030.000000         1030.000000  1030.000000  1030.000000
mean    281.167864           73.895825    54.188350   181.567282
std     104.506364           86.279342    63.997004    21.354219
```

```
min     102.000000            0.000000     0.000000   121.800000
25%     192.375000            0.000000     0.000000   164.900000
50%     272.900000           22.000000     0.000000   185.000000
75%     350.000000          142.950000   118.300000   192.000000
max     540.000000          359.400000   200.100000   247.000000

       Superplasticizer  Coarse Aggregate  Fine Aggregate          Age  \
count       1030.000000       1030.000000     1030.000000  1030.000000
mean           6.204660        972.918932      773.580485    45.662136
std            5.973841         77.753954       80.175980    63.169912
min            0.000000        801.000000      594.000000     1.000000
25%            0.000000        932.000000      730.950000     7.000000
50%            6.400000        968.000000      779.500000    28.000000
75%           10.200000       1029.400000      824.000000    56.000000
max           32.200000       1145.000000      992.600000   365.000000

          Strength
count  1030.000000
mean     35.817961
std      16.705742
min       2.330000
25%      23.710000
50%      34.445000
75%      46.135000
max      82.600000
```

[5]:
```python
concrete_data.isnull().sum()
```

[5]:
```
Cement                0
Blast Furnace Slag    0
Fly Ash               0
Water                 0
Superplasticizer      0
Coarse Aggregate      0
Fine Aggregate        0
Age                   0
Strength              0
dtype: int64
```

The data looks very clean and is ready to be used to build our model.

Split data into predictors and target The target variable in this problem is the concrete sample strength. Therefore, our predictors will be all the other columns.

[6]:
```python
concrete_data_columns = concrete_data.columns
```

```
predictors = concrete_data[concrete_data_columns[concrete_data_columns !=␣
↪'Strength']] # all columns except Strength
target = concrete_data['Strength'] # Strength column
```

Let's do a quick sanity check of the predictors and the target dataframes.

[7]:
```
predictors.head()
```

[7]:
```
   Cement  Blast Furnace Slag  Fly Ash  Water  Superplasticizer  \
0   540.0                 0.0      0.0  162.0               2.5
1   540.0                 0.0      0.0  162.0               2.5
2   332.5               142.5      0.0  228.0               0.0
3   332.5               142.5      0.0  228.0               0.0
4   198.6               132.4      0.0  192.0               0.0

   Coarse Aggregate  Fine Aggregate  Age
0            1040.0           676.0   28
1            1055.0           676.0   28
2             932.0           594.0  270
3             932.0           594.0  365
4             978.4           825.5  360
```

[8]:
```
target.head()
```

[8]:
```
0    79.99
1    61.89
2    40.27
3    41.05
4    44.30
Name: Strength, dtype: float64
```

Finally, the last step is to normalize the data by substracting the mean and dividing by the standard deviation.

[9]:
```
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head()
```

[9]:
```
     Cement  Blast Furnace Slag   Fly Ash     Water  Superplasticizer  \
0  2.476712           -0.856472 -0.846733 -0.916319         -0.620147
1  2.476712           -0.856472 -0.846733 -0.916319         -0.620147
2  0.491187            0.795140 -0.846733  2.174405         -1.038638
3  0.491187            0.795140 -0.846733  2.174405         -1.038638
4 -0.790075            0.678079 -0.846733  0.488555         -1.038638

   Coarse Aggregate  Fine Aggregate       Age
0          0.862735       -1.217079 -0.279597
1          1.055651       -1.217079 -0.279597
2         -0.526262       -2.239829  3.551340
```

```
3          -0.526262        -2.239829  5.055221
4           0.070492         0.647569  4.976069
```

Let's save the number of predictors to *n_cols* since we will need this number when building our network.

[10]:
```
n_cols = predictors_norm.shape[1] # number of predictors
```

0.4 Import Keras

Recall from the videos that Keras normally runs on top of a low-level library such as TensorFlow. This means that to be able to use the Keras library, you will have to install TensorFlow first and when you import the Keras library, it will be explicitly displayed what backend was used to install the Keras library. In CC Labs, we used TensorFlow as the backend to install Keras, so it should clearly print that when we import Keras.

Let's go ahead and import the Keras library

[11]:
```
import keras
```

```
Using TensorFlow backend.
```

As you can see, the TensorFlow backend was used to install the Keras library.

Let's import the rest of the packages from the Keras library that we will need to build our regressoin model.

[12]:
```
from keras.models import Sequential
from keras.layers import Dense
```

0.5 Build a Neural Network

Let's define a function that defines our regression model for us so that we can conveniently call it to create our model.

[13]:
```
# define regression model
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(50, activation='relu', input_shape=(n_cols,)))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(1))

    # compile model
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

The above function create a model that has two hidden layers, each of 50 hidden units.

0.6 Train and Test the Network

Let's call the function now to create our model.

[14]:
```
# build the model
model = regression_model()
```

Next, we will train and test the model at the same time using the *fit* method. We will leave out 30% of the data for validation and we will train the model for 100 epochs.

[15]:
```
# fit the model
model.fit(predictors_norm, target, validation_split=0.3, epochs=100, verbose=2)
```

```
Train on 721 samples, validate on 309 samples
Epoch 1/100
 - 0s - loss: 1660.1377 - val_loss: 1143.0640
Epoch 2/100
 - 0s - loss: 1523.9059 - val_loss: 1011.1052
Epoch 3/100
 - 0s - loss: 1325.1134 - val_loss: 822.5143
Epoch 4/100
 - 0s - loss: 1038.2154 - val_loss: 586.4825
Epoch 5/100
 - 0s - loss: 698.7738 - val_loss: 362.4619
Epoch 6/100
 - 0s - loss: 418.6881 - val_loss: 212.2314
Epoch 7/100
 - 0s - loss: 277.2821 - val_loss: 162.6457
Epoch 8/100
 - 0s - loss: 231.4284 - val_loss: 159.1913
Epoch 9/100
 - 0s - loss: 215.7168 - val_loss: 157.8709
Epoch 10/100
 - 0s - loss: 203.6931 - val_loss: 158.1482
Epoch 11/100
 - 0s - loss: 195.4887 - val_loss: 156.6341
Epoch 12/100
 - 0s - loss: 188.5745 - val_loss: 154.4587
Epoch 13/100
 - 0s - loss: 182.9957 - val_loss: 153.1781
Epoch 14/100
 - 0s - loss: 178.0344 - val_loss: 149.8669
Epoch 15/100
 - 0s - loss: 173.9277 - val_loss: 152.5242
Epoch 16/100
 - 0s - loss: 169.5102 - val_loss: 146.3247
Epoch 17/100
 - 0s - loss: 165.9091 - val_loss: 148.0968
Epoch 18/100
```

```
 - 0s - loss: 162.7200 - val_loss: 145.2803
Epoch 19/100
 - 0s - loss: 159.8348 - val_loss: 146.9721
Epoch 20/100
 - 0s - loss: 157.3678 - val_loss: 143.6480
Epoch 21/100
 - 0s - loss: 154.4429 - val_loss: 145.8840
Epoch 22/100
 - 0s - loss: 152.6101 - val_loss: 142.0781
Epoch 23/100
 - 0s - loss: 151.4637 - val_loss: 141.8954
Epoch 24/100
 - 0s - loss: 148.4784 - val_loss: 142.6425
Epoch 25/100
 - 0s - loss: 146.3285 - val_loss: 139.9731
Epoch 26/100
 - 0s - loss: 145.3244 - val_loss: 140.0204
Epoch 27/100
 - 0s - loss: 143.2450 - val_loss: 138.6028
Epoch 28/100
 - 0s - loss: 141.9220 - val_loss: 139.6912
Epoch 29/100
 - 0s - loss: 140.3962 - val_loss: 140.1303
Epoch 30/100
 - 0s - loss: 139.2788 - val_loss: 139.6737
Epoch 31/100
 - 0s - loss: 138.1343 - val_loss: 137.6235
Epoch 32/100
 - 0s - loss: 136.4452 - val_loss: 139.8918
Epoch 33/100
 - 0s - loss: 135.5935 - val_loss: 137.1772
Epoch 34/100
 - 0s - loss: 133.7596 - val_loss: 137.5810
Epoch 35/100
 - 0s - loss: 132.5286 - val_loss: 140.7972
Epoch 36/100
 - 0s - loss: 131.1030 - val_loss: 139.1057
Epoch 37/100
 - 0s - loss: 130.1083 - val_loss: 137.1844
Epoch 38/100
 - 0s - loss: 128.7180 - val_loss: 137.7035
Epoch 39/100
 - 0s - loss: 127.7545 - val_loss: 139.3261
Epoch 40/100
 - 0s - loss: 126.4812 - val_loss: 135.9571
Epoch 41/100
 - 0s - loss: 125.2417 - val_loss: 137.7308
Epoch 42/100
```

```
 - 0s - loss: 123.8980 - val_loss: 137.2717
Epoch 43/100
 - 0s - loss: 122.8892 - val_loss: 136.2065
Epoch 44/100
 - 0s - loss: 122.1091 - val_loss: 138.2439
Epoch 45/100
 - 0s - loss: 120.9098 - val_loss: 134.9990
Epoch 46/100
 - 0s - loss: 119.9991 - val_loss: 137.9615
Epoch 47/100
 - 0s - loss: 118.3137 - val_loss: 138.8652
Epoch 48/100
 - 0s - loss: 117.0378 - val_loss: 137.3919
Epoch 49/100
 - 0s - loss: 115.5511 - val_loss: 134.1835
Epoch 50/100
 - 0s - loss: 113.8509 - val_loss: 136.2344
Epoch 51/100
 - 0s - loss: 112.2659 - val_loss: 133.5108
Epoch 52/100
 - 0s - loss: 110.2875 - val_loss: 134.0008
Epoch 53/100
 - 0s - loss: 108.5977 - val_loss: 132.3212
Epoch 54/100
 - 0s - loss: 106.4394 - val_loss: 132.3104
Epoch 55/100
 - 0s - loss: 103.9009 - val_loss: 130.4516
Epoch 56/100
 - 0s - loss: 101.3357 - val_loss: 129.0572
Epoch 57/100
 - 0s - loss: 98.8990 - val_loss: 128.3138
Epoch 58/100
 - 0s - loss: 96.9462 - val_loss: 127.7728
Epoch 59/100
 - 0s - loss: 93.2144 - val_loss: 126.1288
Epoch 60/100
 - 0s - loss: 90.3830 - val_loss: 121.7659
Epoch 61/100
 - 0s - loss: 87.9509 - val_loss: 121.4203
Epoch 62/100
 - 0s - loss: 84.7853 - val_loss: 120.0472
Epoch 63/100
 - 0s - loss: 81.6367 - val_loss: 119.1588
Epoch 64/100
 - 0s - loss: 78.5383 - val_loss: 114.0035
Epoch 65/100
 - 0s - loss: 75.4104 - val_loss: 115.2481
Epoch 66/100
```

```
 - 0s - loss: 72.7282 - val_loss: 108.8653
Epoch 67/100
 - 0s - loss: 70.4902 - val_loss: 111.9116
Epoch 68/100
 - 0s - loss: 68.5002 - val_loss: 115.8181
Epoch 69/100
 - 0s - loss: 64.4075 - val_loss: 107.6068
Epoch 70/100
 - 0s - loss: 62.4651 - val_loss: 108.3440
Epoch 71/100
 - 0s - loss: 60.4697 - val_loss: 107.7937
Epoch 72/100
 - 0s - loss: 57.3206 - val_loss: 102.7240
Epoch 73/100
 - 0s - loss: 55.7380 - val_loss: 106.3793
Epoch 74/100
 - 0s - loss: 53.5651 - val_loss: 100.9378
Epoch 75/100
 - 0s - loss: 52.2126 - val_loss: 102.0156
Epoch 76/100
 - 0s - loss: 50.3544 - val_loss: 104.0244
Epoch 77/100
 - 0s - loss: 48.8139 - val_loss: 103.0970
Epoch 78/100
 - 0s - loss: 47.7717 - val_loss: 109.2335
Epoch 79/100
 - 0s - loss: 46.8317 - val_loss: 105.2750
Epoch 80/100
 - 0s - loss: 45.8606 - val_loss: 107.8380
Epoch 81/100
 - 0s - loss: 44.4909 - val_loss: 114.8449
Epoch 82/100
 - 0s - loss: 43.1752 - val_loss: 110.3251
Epoch 83/100
 - 0s - loss: 42.3568 - val_loss: 116.2079
Epoch 84/100
 - 0s - loss: 41.2937 - val_loss: 109.5871
Epoch 85/100
 - 0s - loss: 40.4076 - val_loss: 119.2130
Epoch 86/100
 - 0s - loss: 39.3967 - val_loss: 118.4678
Epoch 87/100
 - 0s - loss: 38.7857 - val_loss: 118.2535
Epoch 88/100
 - 0s - loss: 37.8964 - val_loss: 117.3559
Epoch 89/100
 - 0s - loss: 38.1799 - val_loss: 123.1115
Epoch 90/100
```

```
 - 0s - loss: 37.2709 - val_loss: 123.5457
Epoch 91/100
 - 0s - loss: 36.3797 - val_loss: 128.1173
Epoch 92/100
 - 0s - loss: 35.5836 - val_loss: 123.2614
Epoch 93/100
 - 0s - loss: 34.9860 - val_loss: 128.5807
Epoch 94/100
 - 0s - loss: 34.8328 - val_loss: 125.9789
Epoch 95/100
 - 0s - loss: 34.3698 - val_loss: 132.5730
Epoch 96/100
 - 0s - loss: 34.4769 - val_loss: 128.7075
Epoch 97/100
 - 0s - loss: 34.6661 - val_loss: 119.5304
Epoch 98/100
 - 0s - loss: 33.3136 - val_loss: 126.3475
Epoch 99/100
 - 0s - loss: 33.1318 - val_loss: 126.5016
Epoch 100/100
 - 0s - loss: 32.8780 - val_loss: 129.0572
```

```
[15]: <keras.callbacks.History at 0x7f24b3999860>
```

You can refer to this link to learn about other functions that you can use for prediction or evaluation.

Feel free to vary the following and note what impact each change has on the model's performance:

1. Increase or decreate number of neurons in hidden layers
2. Add more hidden layers
3. Increase number of epochs

0.6.1 Thank you for completing this lab!

This notebook was created by Alex Aklson. I hope you found this lab interesting and educational. Feel free to contact me if you have any questions!

This notebook is part of a course on **Coursera** called *Introduction to Deep Learning & Neural Networks with Keras*. If you accessed this notebook outside the course, you can take this course online by clicking here.