IBM Developer SKILLS NETWORK

[(https://cognitiveclass.ai)](https://cognitiveclass.ai)

# Regression Models with Keras

## 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.

## Table of Contents



## Download and Clean Dataset

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

In [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.

In [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()
```

Out[2]:

| | Cement | Blast Furnace Slag | Fly Ash | Water | Superplasticizer | Coarse Aggregate | Fine Aggregate | Age | Strength |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 540.0 | 0.0 | 0.0 | 162.0 | 2.5 | 1040.0 | 676.0 | 28 | 79.99 |
| **1** | 540.0 | 0.0 | 0.0 | 162.0 | 2.5 | 1055.0 | 676.0 | 28 | 61.89 |
| **2** | 332.5 | 142.5 | 0.0 | 228.0 | 0.0 | 932.0 | 594.0 | 270 | 40.27 |
| **3** | 332.5 | 142.5 | 0.0 | 228.0 | 0.0 | 932.0 | 594.0 | 365 | 41.05 |
| **4** | 198.6 | 132.4 | 0.0 | 192.0 | 0.0 | 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.**

In [3]:

```
concrete_data.shape
```

Out[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.

In [4]:

```
concrete_data.describe()
```

Out[4]:

| | Cement | Blast Furnace Slag | Fly Ash | Water | Superplasticizer | Coarse Aggregate | Aggı |
|---|---|---|---|---|---|---|---|
| **count** | 1030.000000 | 1030.000000 | 1030.000000 | 1030.000000 | 1030.000000 | 1030.000000 | 1030.0 |
| **mean** | 281.167864 | 73.895825 | 54.188350 | 181.567282 | 6.204660 | 972.918932 | 773.5 |
| **std** | 104.506364 | 86.279342 | 63.997004 | 21.354219 | 5.973841 | 77.753954 | 80.1 |
| **min** | 102.000000 | 0.000000 | 0.000000 | 121.800000 | 0.000000 | 801.000000 | 594.0 |
| **25%** | 192.375000 | 0.000000 | 0.000000 | 164.900000 | 0.000000 | 932.000000 | 730.9 |
| **50%** | 272.900000 | 22.000000 | 0.000000 | 185.000000 | 6.400000 | 968.000000 | 779.5 |
| **75%** | 350.000000 | 142.950000 | 118.300000 | 192.000000 | 10.200000 | 1029.400000 | 824.0 |
| **max** | 540.000000 | 359.400000 | 200.100000 | 247.000000 | 32.200000 | 1145.000000 | 992.6 |

In [5]:

```
concrete_data.isnull().sum()
```

Out[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.

In [6]:

```
concrete_data_columns = concrete_data.columns

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

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

In [7]:

```
predictors.head()
```

Out[7]:

| | Cement | Blast Furnace Slag | Fly Ash | Water | Superplasticizer | Coarse Aggregate | Fine Aggregate | Age |
|---|---|---|---|---|---|---|---|---|
| **0** | 540.0 | 0.0 | 0.0 | 162.0 | 2.5 | 1040.0 | 676.0 | 28 |
| **1** | 540.0 | 0.0 | 0.0 | 162.0 | 2.5 | 1055.0 | 676.0 | 28 |
| **2** | 332.5 | 142.5 | 0.0 | 228.0 | 0.0 | 932.0 | 594.0 | 270 |
| **3** | 332.5 | 142.5 | 0.0 | 228.0 | 0.0 | 932.0 | 594.0 | 365 |
| **4** | 198.6 | 132.4 | 0.0 | 192.0 | 0.0 | 978.4 | 825.5 | 360 |

In [8]:

```
target.head()
```

Out[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.

In [9]:

```
predictors_norm = (predictors - predictors.mean()) / predictors.std()
predictors_norm.head()
```

Out[9]:

| | Cement | Blast Furnace Slag | Fly Ash | Water | Superplasticizer | Coarse Aggregate | Fine Aggregate | Age |
|---|---|---|---|---|---|---|---|---|
| **0** | 2.476712 | -0.856472 | -0.846733 | -0.916319 | -0.620147 | 0.862735 | -1.217079 | -0.279597 |
| **1** | 2.476712 | -0.856472 | -0.846733 | -0.916319 | -0.620147 | 1.055651 | -1.217079 | -0.279597 |
| **2** | 0.491187 | 0.795140 | -0.846733 | 2.174405 | -1.038638 | -0.526262 | -2.239829 | 3.551340 |
| **3** | 0.491187 | 0.795140 | -0.846733 | 2.174405 | -1.038638 | -0.526262 | -2.239829 | 5.055221 |
| **4** | -0.790075 | 0.678079 | -0.846733 | 0.488555 | -1.038638 | 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.

In [10]:

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

# 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

In [11]:

```
import keras
```

```
Using TensorFlow backend.
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorfl
ow/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) or
'1type' as a synonym of type is deprecated; in a future version of nump
y, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

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.

In [12]:

```
from keras.models import Sequential
from keras.layers import Dense
```

## 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.

In [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.

## Train and Test the Network

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

In [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.

In [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
 - 1s - loss: 1661.9542 - val_loss: 1171.1349
Epoch 2/100
 - 0s - loss: 1544.0986 - val_loss: 1072.2833
Epoch 3/100
 - 0s - loss: 1350.0741 - val_loss: 908.9197
Epoch 4/100
 - 0s - loss: 1044.0884 - val_loss: 676.1352
Epoch 5/100
 - 0s - loss: 661.6633 - val_loss: 423.1126
Epoch 6/100
 - 0s - loss: 360.7913 - val_loss: 256.2565
Epoch 7/100
 - 0s - loss: 258.6727 - val_loss: 194.3958
Epoch 8/100
 - 0s - loss: 231.9348 - val_loss: 181.2329
Epoch 9/100
 - 0s - loss: 216.1197 - val_loss: 175.4686
Epoch 10/100
 - 0s - loss: 203.6296 - val_loss: 170.0406
Epoch 11/100
 - 0s - loss: 195.2363 - val_loss: 162.6223
Epoch 12/100
 - 0s - loss: 187.8564 - val_loss: 163.6579
Epoch 13/100
 - 0s - loss: 182.3975 - val_loss: 157.7677
Epoch 14/100
 - 0s - loss: 176.7468 - val_loss: 157.8341
Epoch 15/100
 - 0s - loss: 172.8861 - val_loss: 157.4784
Epoch 16/100
 - 0s - loss: 168.3367 - val_loss: 151.4854
Epoch 17/100
 - 0s - loss: 165.0218 - val_loss: 150.2919
Epoch 18/100
 - 0s - loss: 161.9014 - val_loss: 148.6780
Epoch 19/100
 - 0s - loss: 159.2922 - val_loss: 146.4039
Epoch 20/100
 - 0s - loss: 157.3730 - val_loss: 146.7764
Epoch 21/100
 - 0s - loss: 154.0957 - val_loss: 147.9762
Epoch 22/100
 - 0s - loss: 152.3168 - val_loss: 144.9887
Epoch 23/100
 - 0s - loss: 150.0433 - val_loss: 144.0366
Epoch 24/100
 - 0s - loss: 148.2346 - val_loss: 143.9639
Epoch 25/100
 - 0s - loss: 146.2840 - val_loss: 142.8641
Epoch 26/100
 - 0s - loss: 145.0572 - val_loss: 144.5669
Epoch 27/100
 - 0s - loss: 142.5175 - val_loss: 143.3486
Epoch 28/100
 - 0s - loss: 140.8053 - val_loss: 140.7602
```

```
Epoch 29/100
 - 0s - loss: 139.7074 - val_loss: 141.7203
Epoch 30/100
 - 0s - loss: 138.3224 - val_loss: 142.2949
Epoch 31/100
 - 0s - loss: 136.3011 - val_loss: 140.7438
Epoch 32/100
 - 0s - loss: 134.9698 - val_loss: 141.1045
Epoch 33/100
 - 0s - loss: 133.6137 - val_loss: 141.8782
Epoch 34/100
 - 0s - loss: 132.9054 - val_loss: 142.9442
Epoch 35/100
 - 0s - loss: 130.6358 - val_loss: 141.1838
Epoch 36/100
 - 0s - loss: 129.1508 - val_loss: 140.8346
Epoch 37/100
 - 0s - loss: 127.6293 - val_loss: 139.8212
Epoch 38/100
 - 0s - loss: 125.8805 - val_loss: 139.4211
Epoch 39/100
 - 0s - loss: 124.7260 - val_loss: 141.6114
Epoch 40/100
 - 0s - loss: 123.5292 - val_loss: 139.0252
Epoch 41/100
 - 0s - loss: 121.8977 - val_loss: 139.1942
Epoch 42/100
 - 0s - loss: 120.3779 - val_loss: 138.1366
Epoch 43/100
 - 0s - loss: 119.1327 - val_loss: 138.6886
Epoch 44/100
 - 0s - loss: 117.5295 - val_loss: 138.5328
Epoch 45/100
 - 0s - loss: 115.6472 - val_loss: 138.7706
Epoch 46/100
 - 0s - loss: 114.7165 - val_loss: 134.9039
Epoch 47/100
 - 0s - loss: 112.4688 - val_loss: 139.5132
Epoch 48/100
 - 0s - loss: 109.8030 - val_loss: 134.0823
Epoch 49/100
 - 0s - loss: 107.4404 - val_loss: 133.5114
Epoch 50/100
 - 0s - loss: 105.3394 - val_loss: 132.1645
Epoch 51/100
 - 0s - loss: 102.8611 - val_loss: 133.2495
Epoch 52/100
 - 0s - loss: 99.7087 - val_loss: 130.6321
Epoch 53/100
 - 0s - loss: 97.1214 - val_loss: 129.0231
Epoch 54/100
 - 0s - loss: 93.7959 - val_loss: 126.5652
Epoch 55/100
 - 0s - loss: 91.1826 - val_loss: 126.6099
Epoch 56/100
 - 0s - loss: 87.7261 - val_loss: 126.5160
Epoch 57/100
```

```
 - 0s - loss: 83.8065 - val_loss: 119.9674
Epoch 58/100
 - 0s - loss: 81.3501 - val_loss: 118.0468
Epoch 59/100
 - 0s - loss: 77.3190 - val_loss: 118.0448
Epoch 60/100
 - 0s - loss: 74.1334 - val_loss: 119.7944
Epoch 61/100
 - 0s - loss: 71.7401 - val_loss: 112.8265
Epoch 62/100
 - 0s - loss: 68.1371 - val_loss: 111.5210
Epoch 63/100
 - 0s - loss: 65.1932 - val_loss: 112.1519
Epoch 64/100
 - 0s - loss: 63.3672 - val_loss: 107.5093
Epoch 65/100
 - 0s - loss: 61.4357 - val_loss: 106.7303
Epoch 66/100
 - 0s - loss: 58.7450 - val_loss: 108.6653
Epoch 67/100
 - 0s - loss: 56.4865 - val_loss: 112.2597
Epoch 68/100
 - 0s - loss: 54.8916 - val_loss: 108.8864
Epoch 69/100
 - 0s - loss: 53.3470 - val_loss: 109.0643
Epoch 70/100
 - 0s - loss: 51.4928 - val_loss: 111.5028
Epoch 71/100
 - 0s - loss: 49.9468 - val_loss: 112.3113
Epoch 72/100
 - 0s - loss: 48.9900 - val_loss: 111.9038
Epoch 73/100
 - 0s - loss: 47.4502 - val_loss: 121.2940
Epoch 74/100
 - 0s - loss: 45.8812 - val_loss: 109.9098
Epoch 75/100
 - 0s - loss: 44.4248 - val_loss: 126.2543
Epoch 76/100
 - 0s - loss: 44.0906 - val_loss: 119.0120
Epoch 77/100
 - 0s - loss: 43.2289 - val_loss: 125.0030
Epoch 78/100
 - 0s - loss: 42.2393 - val_loss: 121.3367
Epoch 79/100
 - 0s - loss: 41.5181 - val_loss: 132.1536
Epoch 80/100
 - 0s - loss: 40.8834 - val_loss: 122.1942
Epoch 81/100
 - 0s - loss: 39.9881 - val_loss: 125.0803
Epoch 82/100
 - 0s - loss: 39.3678 - val_loss: 145.5577
Epoch 83/100
 - 0s - loss: 39.6371 - val_loss: 128.6204
Epoch 84/100
 - 0s - loss: 38.1649 - val_loss: 137.7343
Epoch 85/100
 - 0s - loss: 37.9042 - val_loss: 134.3885
```

```
Epoch 86/100
 - 0s - loss: 36.8736 - val_loss: 133.0377
Epoch 87/100
 - 0s - loss: 36.5789 - val_loss: 137.7313
Epoch 88/100
 - 0s - loss: 36.5297 - val_loss: 132.1596
Epoch 89/100
 - 0s - loss: 36.1719 - val_loss: 137.5838
Epoch 90/100
 - 0s - loss: 35.2861 - val_loss: 138.0206
Epoch 91/100
 - 0s - loss: 35.2967 - val_loss: 139.0921
Epoch 92/100
 - 0s - loss: 34.7275 - val_loss: 137.7480
Epoch 93/100
 - 0s - loss: 34.2688 - val_loss: 135.8431
Epoch 94/100
 - 0s - loss: 34.9694 - val_loss: 135.8543
Epoch 95/100
 - 0s - loss: 33.9289 - val_loss: 135.9575
Epoch 96/100
 - 0s - loss: 33.6405 - val_loss: 143.9162
Epoch 97/100
 - 0s - loss: 33.0528 - val_loss: 142.0255
Epoch 98/100
 - 0s - loss: 33.2523 - val_loss: 136.9596
Epoch 99/100
 - 0s - loss: 32.9750 - val_loss: 151.1339
Epoch 100/100
 - 0s - loss: 32.7311 - val_loss: 136.8290
```

Out[15]:

```
<keras.callbacks.History at 0x7f3e7adb47f0>
```

**You can refer to this [link](https://keras.io/models/sequential/) 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

## Thank you for completing this lab!

This notebook was created by [Alex Aklson (https://www.linkedin.com/in/aklson/)](https://www.linkedin.com/in/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 (https://cocl.us/DL0101EN_Coursera_Week3_LAB1).

---