

Build a Regression Model

by Mei Chiao Lin

Jun/3rd/2020

Part B -- One hidden layer, *normalized data*, 50 epochs

In [1]:

```
import keras
```

Using TensorFlow backend.

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/tensorflow/python/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, 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/tensorflow/python/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, 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/tensorflow/python/framework/dtypes.py:521: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, 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/tensorflow/python/framework/dtypes.py:522: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, 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/tensorflow/python/framework/dtypes.py:523: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, 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/tensorflow/python/framework/dtypes.py:528: FutureWarning: Passing (type, 1) or 'ltype' as a synonym of type is deprecated; in a future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
```

```
np_resource = np.dtype [("resource", np.ubyte, 1)]
```

In [2]:

```
import pandas as pd
import numpy as np
```

In [3]:

```
from sklearn.model_selection import train_test_split
```

In [4]:

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

Out[4]:

	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

In [5]:

```
concrete_data.shape
```

Out[5]:

(1030, 9)

In [6]:

```
concrete_data.describe()
```

Out[6]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Aggr
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 [7]:

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

Out[7]:

```
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
```

In [8]:

```
concrete_data_columns = concrete_data.columns
X = concrete_data[concrete_data_columns[concrete_data_columns != 'Strength']] # all
columns except Strength
y = concrete_data['Strength'] # Strength column
n_cols=X.shape[1]
```

Normalization

In [9]:

```
#Normalization part by mean and standard deviation
X_nor = (X-np.mean(X))/np.std(X)
y_nor = (y-np.mean(y))/np.std(y)
```

Splitting the data

In [10]:

```
#Split the data into training dataset and testing dataset with 30% test dataset
X_train_nor, X_test_nor, y_train_nor, y_test_nor = train_test_split(X_nor, y_nor, t
est_size=0.3)
```

In [11]:

```
X_nor.head()
```

Out[11]:

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age
0	2.477915	-0.856888	-0.847144	-0.916764	-0.620448	0.863154	-1.217670	-0.279733
1	2.477915	-0.856888	-0.847144	-0.916764	-0.620448	1.056164	-1.217670	-0.279733
2	0.491425	0.795526	-0.847144	2.175461	-1.039143	-0.526517	-2.240917	3.553066
3	0.491425	0.795526	-0.847144	2.175461	-1.039143	-0.526517	-2.240917	5.057677
4	-0.790459	0.678408	-0.847144	0.488793	-1.039143	0.070527	0.647884	4.978487

In [12]:

```
y_nor.head()
```

Out[12]:

```
0    2.645408
1    1.561421
2    0.266627
3    0.313340
4    0.507979
Name: Strength, dtype: float64
```

Building the model

In [13]:

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

In [14]:

```
#One hidden layer with 10 nodes and relu function
#adam optimizer and mean_squared_error as loss function
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(10, activation='relu', input_shape=(n_cols,)))
    model.add(Dense(1))

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

Training

In [15]:

```
#build the model
model = regression_model()
```

In [16]:

```
#fitting data to the model with 50 epoch  
model.fit(X_train_nor, y_train_nor, epochs=50)
```

Epoch 1/50
721/721 [=====] - 3s 4ms/step - loss: 1.6053A:
2s - los

Epoch 2/50
721/721 [=====] - 1s 2ms/step - loss: 1.2588

Epoch 3/50
721/721 [=====] - 1s 1ms/step - loss: 1.0351

Epoch 4/50
721/721 [=====] - 1s 2ms/step - loss: 0.8721

Epoch 5/50
721/721 [=====] - 1s 1ms/step - loss: 0.7558

Epoch 6/50
721/721 [=====] - 2s 2ms/step - loss: 0.6752

Epoch 7/50
721/721 [=====] - 1s 2ms/step - loss: 0.6114

Epoch 8/50
721/721 [=====] - 2s 2ms/step - loss: 0.5601

Epoch 9/50
721/721 [=====] - 1s 2ms/step - loss: 0.5193

Epoch 10/50
721/721 [=====] - 2s 2ms/step - loss: 0.4819

Epoch 11/50
721/721 [=====] - 1s 2ms/step - loss: 0.4521

Epoch 12/50
721/721 [=====] - 1s 2ms/step - loss: 0.4283

Epoch 13/50
721/721 [=====] - 1s 2ms/step - loss: 0.4076

Epoch 14/50
721/721 [=====] - 1s 2ms/step - loss: 0.3922

Epoch 15/50
721/721 [=====] - 1s 2ms/step - loss: 0.3787

Epoch 16/50
721/721 [=====] - 1s 1ms/step - loss: 0.3671

Epoch 17/50
721/721 [=====] - 1s 2ms/step - loss: 0.3575

Epoch 18/50
721/721 [=====] - 1s 2ms/step - loss: 0.3484

Epoch 19/50
721/721 [=====] - 2s 2ms/step - loss: 0.3404

Epoch 20/50
721/721 [=====] - 1s 2ms/step - loss: 0.3327

Epoch 21/50
721/721 [=====] - 1s 2ms/step - loss: 0.3262

Epoch 22/50
721/721 [=====] - 1s 2ms/step - loss: 0.3190

Epoch 23/50
721/721 [=====] - 2s 2ms/step - loss: 0.3131A:
1s - l - ETA: 0s - loss

Epoch 24/50
721/721 [=====] - 1s 2ms/step - loss: 0.3071

Epoch 25/50
721/721 [=====] - 1s 2ms/step - loss: 0.3010

Epoch 26/50
721/721 [=====] - 1s 1ms/step - loss: 0.2963

Epoch 27/50
721/721 [=====] - 1s 2ms/step - loss: 0.2920A:
0s - loss

```

Epoch 28/50
721/721 [=====] - 1s 1ms/step - loss: 0.2861
Epoch 29/50
721/721 [=====] - 1s 2ms/step - loss: 0.2806
Epoch 30/50
721/721 [=====] - 1s 1ms/step - loss: 0.2765
Epoch 31/50
721/721 [=====] - 1s 2ms/step - loss: 0.2707
Epoch 32/50
721/721 [=====] - 1s 1ms/step - loss: 0.2661
Epoch 33/50
721/721 [=====] - 1s 2ms/step - loss: 0.2613
Epoch 34/50
721/721 [=====] - 1s 2ms/step - loss: 0.2568
Epoch 35/50
721/721 [=====] - 1s 2ms/step - loss: 0.2524
Epoch 36/50
721/721 [=====] - 1s 2ms/step - loss: 0.2477A:
2s
Epoch 37/50
721/721 [=====] - 1s 1ms/step - loss: 0.2434
Epoch 38/50
721/721 [=====] - 1s 2ms/step - loss: 0.2394
Epoch 39/50
721/721 [=====] - 2s 2ms/step - loss: 0.2356
Epoch 40/50
721/721 [=====] - 2s 2ms/step - loss: 0.2318
Epoch 41/50
721/721 [=====] - 1s 1ms/step - loss: 0.2277
Epoch 42/50
721/721 [=====] - 1s 941us/step - loss: 0.2243
Epoch 43/50
721/721 [=====] - 1s 2ms/step - loss: 0.2207
Epoch 44/50
721/721 [=====] - 1s 2ms/step - loss: 0.2173
Epoch 45/50
721/721 [=====] - ETA: 0s - loss: 0.214 - 1s 1
ms/step - loss: 0.2138
Epoch 46/50
721/721 [=====] - 1s 1ms/step - loss: 0.2108
Epoch 47/50
721/721 [=====] - 2s 2ms/step - loss: 0.2075
Epoch 48/50
721/721 [=====] - 2s 2ms/step - loss: 0.2038
Epoch 49/50
721/721 [=====] - 1s 2ms/step - loss: 0.2009
Epoch 50/50
721/721 [=====] - 2s 2ms/step - loss: 0.1975

```

Out[16]:

```
<keras.callbacks.History at 0x7f79b69c6b70>
```


In [17]:

```
#evaluate the model
evaluated_score = model.evaluate(X_test_nor, y_test_nor, verbose=1)

309/309 [=====] - 0s 1ms/step
```

In [18]:

```
y_predict=model.predict(X_test_nor)
```

In [19]:

```
from sklearn.metrics import mean_squared_error
```

In [20]:

```
squared_error_score = mean_squared_error(y_test_nor, y_predict)
```

Repeat 50 times

In [21]:

```
#Train and evaluate the model for 50 times.
error_score_nor=[]
for i in range(50):
    model.fit(X_train_nor, y_train_nor, epochs=50, verbose=0)
    y_predict_nor=model.predict(X_test_nor)
    error_score_nor.append(mean_squared_error(y_test_nor,y_predict_nor))
```

In [22]:

```
Mean_nor = np.mean(error_score_nor)
Std_nor = np.std(error_score_nor)
```

In [25]:

```
print('The mean of mean_squared_error is : 64.019, while in normalized data is {:.3f}\n\nThe standard deviation of mean_squared_error is : 21.461, while in normalized data is {:.3f}'.format(Mean_nor, Std_nor))
```

The mean of mean_squared_error is : 64.019, while in normalized data is 0.134

The standard deviation of mean_squared_error is : 21.461, while in normalized data is 0.005

How does the mean squared of error compared to Part A ?

The range of loss value in Part A is big, but in Part B is between 0 and 1.

For the practical reason, the result from part B is more convenient.