Project Deep Learning. Developing Models for Regression & Classification.

Index

• Main objective of the analysis

The main objective of the analysis will be to see the application of Deep learning both for Regression & Classification using Keras and 2 different datasets.

Case a. For regression, I will use a of **Concrete Strength** and

Case b. for Classification I will use the dataset **MNIST**.

The goal is to build a model of deep learning neural network using Keras to define

- *the number of hidden layers,*
- *the number of nodes and*
- *the type of activation,*

*and analyze how the configuration can generate different results, evaluating the **accuracy**.*

Case a. Regression with a Dataset of Concrete Strength

- **Brief description of the data set & summary of data exploration**.

We'll see first the features of the dataset and the features that actuate as predictors.

Download and Clean Dataset

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

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

As required for the project, 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

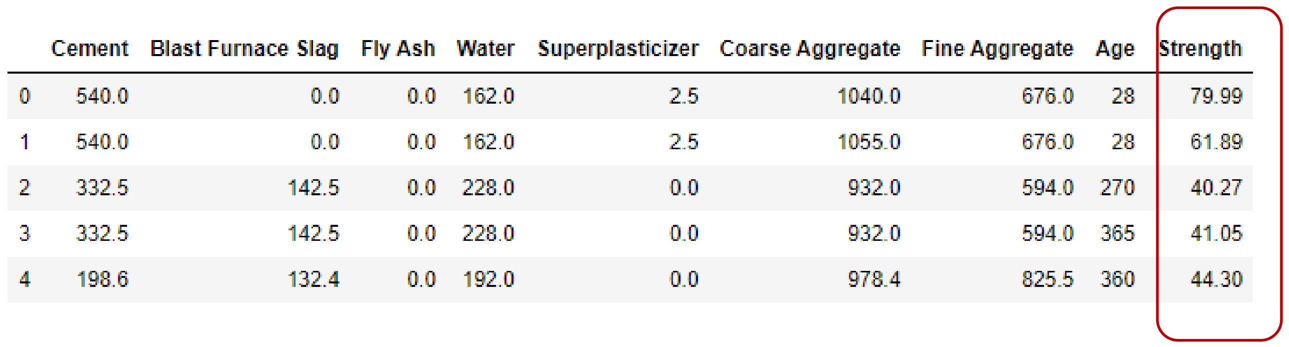
5. Superplasticizer

6. Coarse Aggregate

7. Fine Aggregate

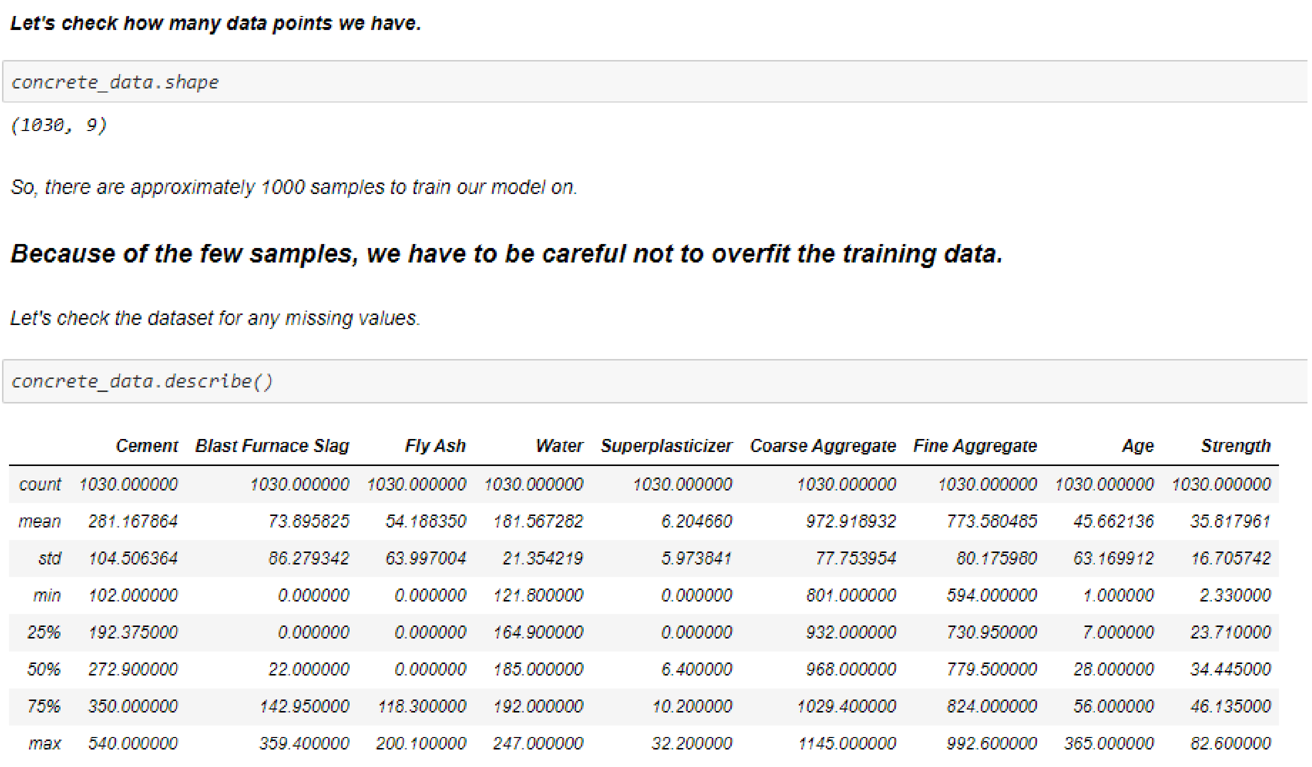
Let's download the data and read it into a *pandas* dataframe.

```
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()
```



	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.

```
concrete_data.shape
```

```
(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.

```
concrete_data.describe()
```

	Cement	Blast Furnace Slag	Fly Ash	Water	Superplasticizer	Coarse Aggregate	Fine Aggregate	Age	Strength
count	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000
mean	281.167864	73.895825	54.188350	181.567282	6.204660	972.918932	773.580485	45.662136	35.817961
std	104.506364	86.279342	63.997004	21.354219	5.973841	77.753954	80.175980	63.169912	16.705742
min	102.000000	0.000000	0.000000	121.800000	0.000000	801.000000	594.000000	1.000000	2.330000
25%	192.375000	0.000000	0.000000	164.900000	0.000000	932.000000	730.950000	7.000000	23.710000
50%	272.900000	22.000000	0.000000	185.000000	6.400000	968.000000	779.500000	28.000000	34.445000
75%	350.000000	142.950000	118.300000	192.000000	10.200000	1029.400000	824.000000	56.000000	46.135000
max	540.000000	359.400000	200.100000	247.000000	32.200000	1145.000000	992.600000	365.000000	82.600000

And to see missing values for features we execute

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

```
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.

- **Summary of training at least three different classifier models,**

Then, **we split the dataset into Predictors & Target.**

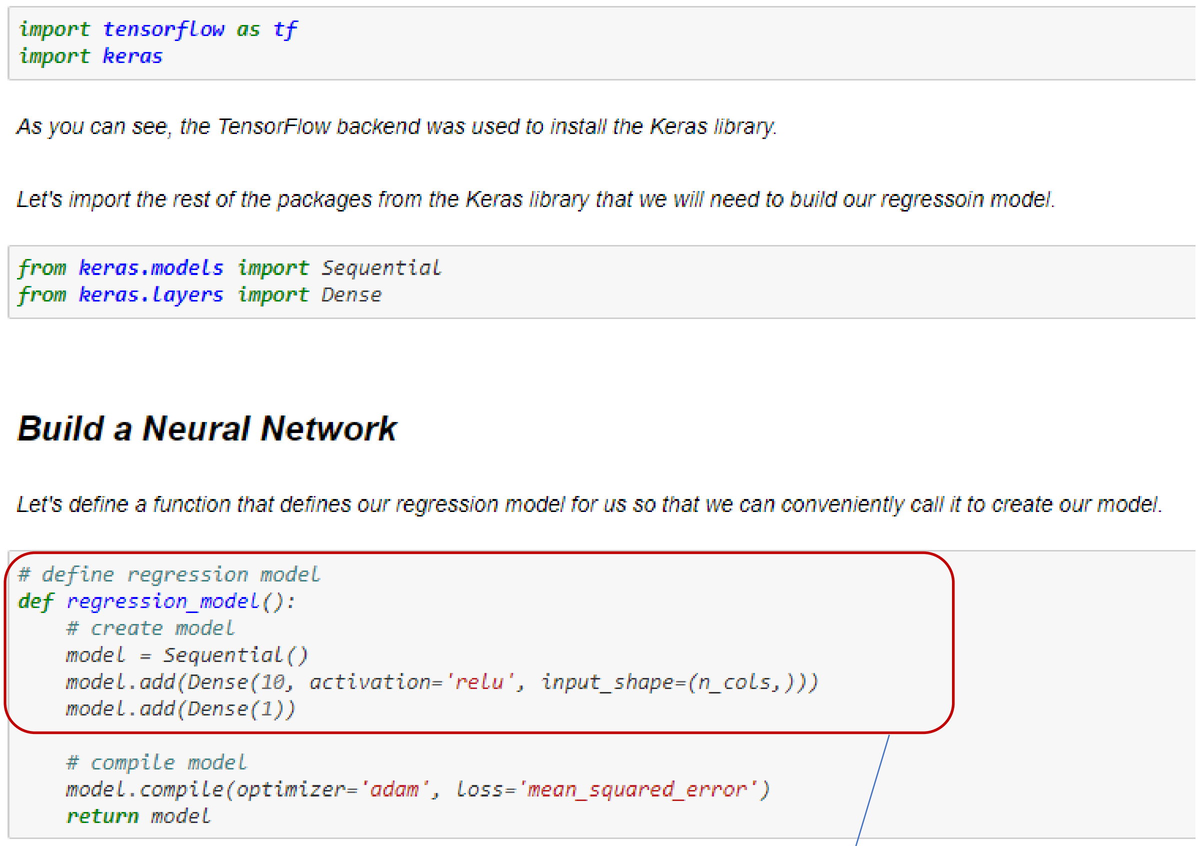
The target variable in this problem is the concrete sample strength. Therefore, our predictors will be all the other columns.

In [6]:

```
concrete_data_columns = concrete_data.columns

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

And we **import Keras and Tensorflow and define the model**



```
import tensorflow as tf
import keras
```

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.

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

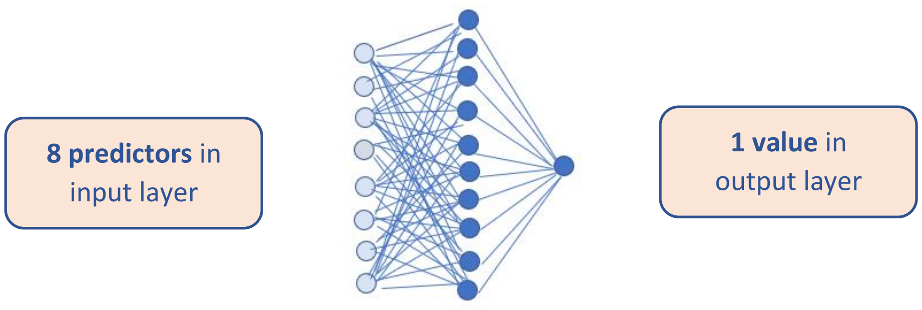
Build a Neural Network

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

```
# define regression model
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
```

The above function create a model that has ONE hidden layer, of 10 nodes.



And now we **Train & Test the Network**

```
# build the model

model = regression_model()
```

Next, we will train and test the model at the same time using the fit method.

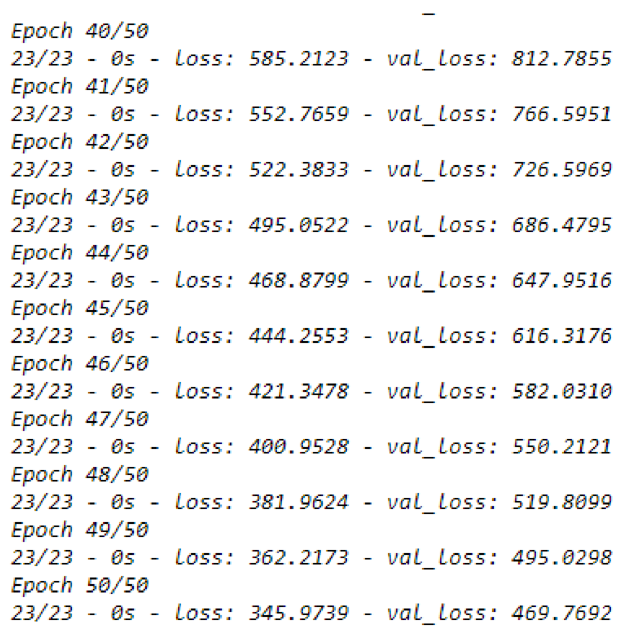
*We will leave out **30%** of the **data for validation** and we will train the model for **50 epochs**.*

```
# fit the model

model.fit(predictors, target, validation_split=0.3, epochs=50, verbose=2)
```

Obtaining for the last 10 epochs the following results for **loss of Training set (**loss**) and Test Set (**val-loss**).**

The value of loss is the *mean squared error*, as it is defined in the model.



```
Epoch 40/50
23/23 - 0s - loss: 585.2123 - val_loss: 812.7855
Epoch 41/50
23/23 - 0s - loss: 552.7659 - val_loss: 766.5951
Epoch 42/50
23/23 - 0s - loss: 522.3833 - val_loss: 726.5969
Epoch 43/50
23/23 - 0s - loss: 495.0522 - val_loss: 686.4795
Epoch 44/50
23/23 - 0s - loss: 468.8799 - val_loss: 647.9516
Epoch 45/50
23/23 - 0s - loss: 444.2553 - val_loss: 616.3176
Epoch 46/50
23/23 - 0s - loss: 421.3478 - val_loss: 582.0310
Epoch 47/50
23/23 - 0s - loss: 400.9528 - val_loss: 550.2121
Epoch 48/50
23/23 - 0s - loss: 381.9624 - val_loss: 519.8099
Epoch 49/50
23/23 - 0s - loss: 362.2173 - val_loss: 495.0298
Epoch 50/50
23/23 - 0s - loss: 345.9739 - val_loss: 469.7692
```

And repeating the process several iterations we can observe **than from 4th or 5th iteration the results for Test set begin to increase while Training Set continue decreasing**, showing that there is some **overfit** probably related to the fact that the number of rows is small (only 1000).

We can consider 3 more options in order to see how is affecting:

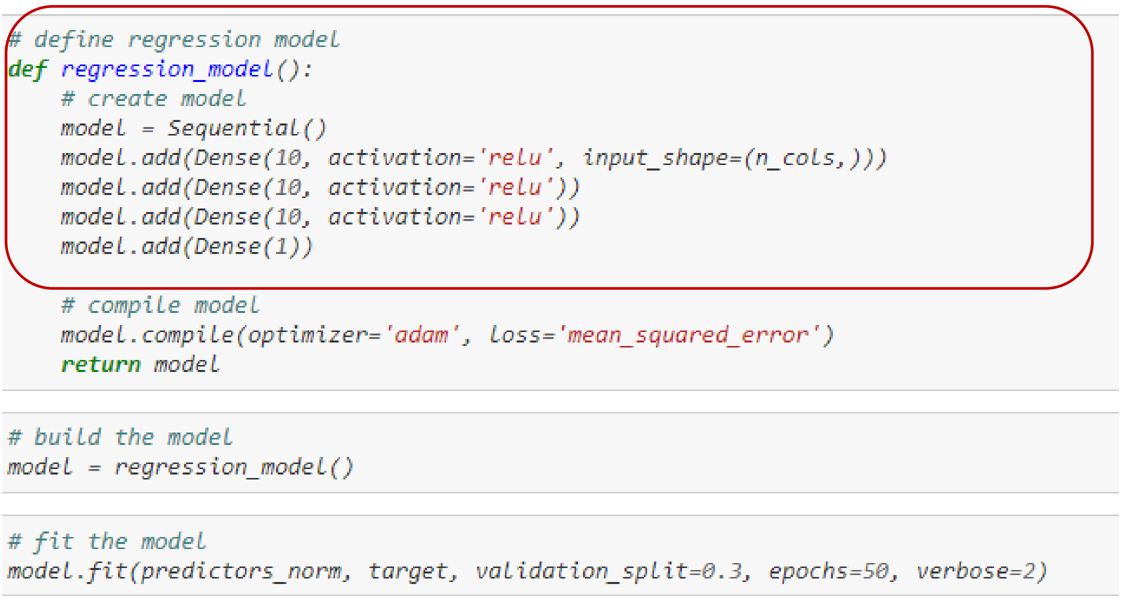
a) We can see what happens **if we normalize data**
b) Considering a different number of **epochs**
c) Including **more hidden layers**

a. We can see that it is convenient to normalize data as we appreciate a reduction of loss. To do it, we apply

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

b. We see that 100 epochs can obtain a lower value of loss for Training Set, but for Test Set is showing again some overfit.

c. The **definition of the model for 3 hidden layers** will be



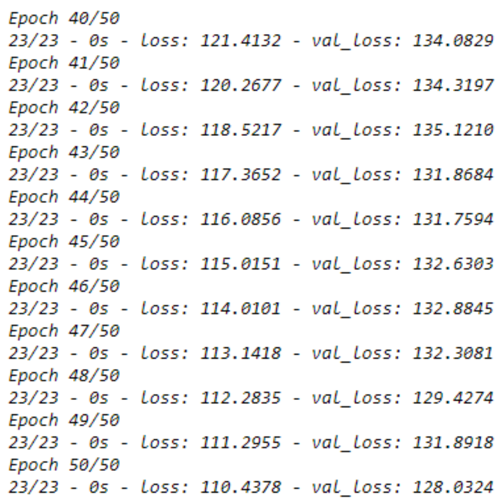
```
# define regression model
def regression_model():
    # create model
    model = Sequential()
    model.add(Dense(10, activation='relu', input_shape=(n_cols,)))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1))

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

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

```
# fit the model
model.fit(predictors_norm, target, validation_split=0.3, epochs=50, verbose=2)
```

We can see that "loss" and "val_loss" are lower than for initial model with one hidden layer.



```
Epoch 40/50
23/23 - 0s - loss: 121.4132 - val_loss: 134.0829
Epoch 41/50
23/23 - 0s - loss: 120.2677 - val_loss: 134.3197
Epoch 42/50
23/23 - 0s - loss: 118.5217 - val_loss: 135.1210
Epoch 43/50
23/23 - 0s - loss: 117.3652 - val_loss: 131.8684
Epoch 44/50
23/23 - 0s - loss: 116.0856 - val_loss: 131.7594
Epoch 45/50
23/23 - 0s - loss: 115.0151 - val_loss: 132.6303
Epoch 46/50
23/23 - 0s - loss: 114.0101 - val_loss: 132.8845
Epoch 47/50
23/23 - 0s - loss: 113.1418 - val_loss: 132.3081
Epoch 48/50
23/23 - 0s - loss: 112.2835 - val_loss: 129.4274
Epoch 49/50
23/23 - 0s - loss: 111.2955 - val_loss: 131.8918
Epoch 50/50
23/23 - 0s - loss: 110.4378 - val_loss: 128.0324
```

- **Summary Key Findings and Insights & Suggestions for next steps**

Then we could conclude that the options a. (**normalizing data**) & c.(**model with 3 hidden layers**) could be convenient for the model as we can obtain a reduction in loss (*mean squared error*) for Training Set although there is **overfit** in the results in Test Set if we run a high number of iterations.

Some options for **next steps** could be:

- See how work **networks having more nodes** in each layer (for example **50**)
- Define the model with a **different type of "activation"** instead "**relu"**.
- Fit the model with a different value of **validation split** (for example 0.2 or 0.4 instead **0.3**)

Case b. for Classification I will use the dataset MNIST.

- ### Brief description of the data set & summary of data exploration.

In this project I will use the popular MNIST dataset. A dataset of images.

The **MNIST database**, short for Modified National Institute of Standards and Technology database, is a large database of handwritten digits that is commonly used for training various image processing systems.

The MNIST database contains 60,000 training images and 10,000 testing images of digits written by high school students and employees of the United States Census Bureau.

Now, **we import the packages**, and we'll see the shape and some sample

- Let's start by importing Keras and some of its modules. to_categorical has to be imported from tensorflow.keras as it is integrated.

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import to_categorical
```

- Since we are dealing with images, let's also import the Matplotlib scripting layer in order to view the images.

```
import matplotlib.pyplot as plt
```

- So, let's load the MNIST dataset from the Keras library. The dataset is readily divided into a training set and a test set

```
# import the data
from keras.datasets import mnist
# read the data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

- Now we'll confirm the number of images in each set. According to the dataset's documentation, we should have 60000 images in X_train and 10000 images in the X_test.

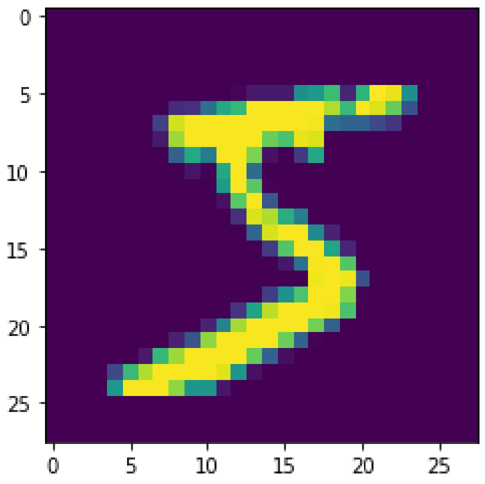
```
X_train.shape
```

```
Out: (60000, 28, 28)
```

Let's visualize the first image in the training set using Matplotlib's scripting layer.

```
plt.imshow(X_train[0])
```

```
<matplotlib.image.AxesImage at 0x2892a1d2488>
```



- With conventional neural networks, we cannot feed in the image as input as is. So we need to flatten the images into one-dimensional vectors, each of size 1 x (28 x 28) = 1 x 784.

```
num_pixels = X_train.shape[1] * X_train.shape[2]
# find size of one-dimensional vector

X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

- Since pixel values can range from 0 to 255, let's normalize the vectors to be between 0 and 1.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

- Finally, before we start building our model, for classification we need to divide our target variable into categories. We use the **to_categorical** function from the Keras Utilities package.

```
# one hot encode outputs
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

num_classes = y_test.shape[1]
print(num_classes)
```

```
Out:  10
```

- **Summary of training classifier model**

Build a Neural Network

```
# define classification model
def classification_model():
    # create model
    model = Sequential()
    model.add(Dense(num_pixels, activation='relu', input_shape=(num_pixel
s,)))
    model.add(Dense(100, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))


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

- In this case, we are considering 2 hidden layers. The first one with “num_pixels” nodes and the second one with 100 nodes.
- The output has “**num_classes**” nodes, that it’s **10**.

Train and Test the Network

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

# fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10, verb
ose=2)

# evaluate the model
scores = model.evaluate(X_test, y_test, verbose=0)
```

```
Epoch 1/10
1875/1875 - 15s - loss: 1.0913 - accuracy: 0.8364 - val_loss: 0.3386 - val_accuracy: 0.9170
Epoch 2/10
1875/1875 - 14s - loss: 0.2635 - accuracy: 0.9304 - val_loss: 0.2071 - val_accuracy: 0.9502
Epoch 3/10
1875/1875 - 15s - loss: 0.1806 - accuracy: 0.9519 - val_loss: 0.2251 - val_accuracy: 0.9480
Epoch 4/10
1875/1875 - 14s - loss: 0.1506 - accuracy: 0.9600 - val_loss: 0.1695 - val_accuracy: 0.9565
Epoch 5/10
1875/1875 - 14s - loss: 0.1251 - accuracy: 0.9667 - val_loss: 0.1365 - val_accuracy: 0.9664
Epoch 6/10
1875/1875 - 14s - loss: 0.1134 - accuracy: 0.9712 - val_loss: 0.1552 - val_accuracy: 0.9603
Epoch 7/10
1875/1875 - 14s - loss: 0.0987 - accuracy: 0.9745 - val_loss: 0.1243 - val_accuracy: 0.9727
```

```
Epoch 8/10
1875/1875 - 14s - loss: 0.0908 - accuracy: 0.9782 - val_loss: 0.1318 - val_accuracy: 0.9722
Epoch 9/10
1875/1875 - 15s - loss: 0.0841 - accuracy: 0.9796 - val_loss: 0.1710 - val_accuracy: 0.9709
Epoch 10/10
1875/1875 - 15s - loss: 0.0883 - accuracy: 0.9793 - val_loss: 0.1544 - val_accuracy: 0.9706
```

- Let's print the accuracy and the corresponding error.

```
Print('Accuracy : {}% \n Error : {}'.format(scores[1], 1 - scores[1]))
```

```
Accuracy: 0.9783999919891357%
 Error: 0.021600008010864258
```

Indicating that the model has a good accuracy for this dataset.

Summary Key Findings and Insights & Suggestions for next steps

- We've seen how **it is possible to generate a good model for a dataset as MNIST obtaining a very good accuracy based in deep learning with a relatively basic configuration of only 2 hidden layers** (***although having a high number of nodes***) of conventional neural network.

- The **optimizer** in the model has been "**adam**", loss is calculated based on "categorical crossentropy", and **fit has been done with 10 epochs.**

- **About next steps,** could be possible to check some different configuration of parameters and model, but for this particular dataset **the most interesting suggestion would be to use CNN and RNN types as it is possible to obtain better results for images.**