Handwritten Character Recognition (MLP + Crossentropy)

By
Avishek Paul

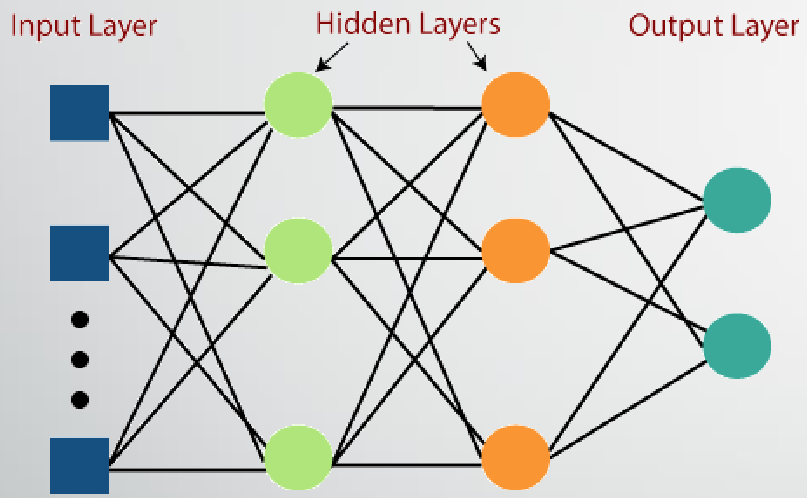
(B. Tech, CSE, NIT Meghalaya)

Handwriting Recognition

Handwriting recognition (HWR), also known as Handwritten Text Recognition (HTR), is the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens and other devices.

In this project, we will recognize handwritten characters, i.e., English alphabets from A-Z. This we are going to achieve by modeling a Multi-Layer Perceptron (MLP) neural network with Crossentropy as the loss function that will have to be trained over a dataset containing images of alphabets.

What is MLP?



A Multi-Layer Perceptron (MLP) is a class of feedforward artificial neural network (ANN). The term MLP is used ambiguously, sometimes loosely to any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons (with threshold activation). Multilayer perceptrons are sometimes colloquially referred to as "vanilla" neural networks, especially when they have a single hidden layer

An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function.

MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron.

Project Prerequisites

Below are the prerequisites for this project:

- Python (3.7)
- IDE – Jupyter Notebook (Google Colab)

Frameworks used:

- NumPy (1.19.5)
- Pandas (1.1.5)
- Keras (2.5.0)
- TensorFlow (Keras uses TensorFlow in backend and for some image pre-processing) (2.5.0)
- Matplotlib (3.2.2)
- Seaborn (0.11.1)

Dataset Used

The dataset used is the A-Z Handwritten Alphabets Dataset from Kaggle. The dataset for this project contains 372450 images of alphabets of 28×28 pixels, each alphabet in the image is center fitted to 2020 pixel box, all present in the form of a CSV file.

The images are taken from NIST(https://www.nist.gov/srd/nist-special-database-19) and NMIST large dataset and few other sources which were then formatted as mentioned above.

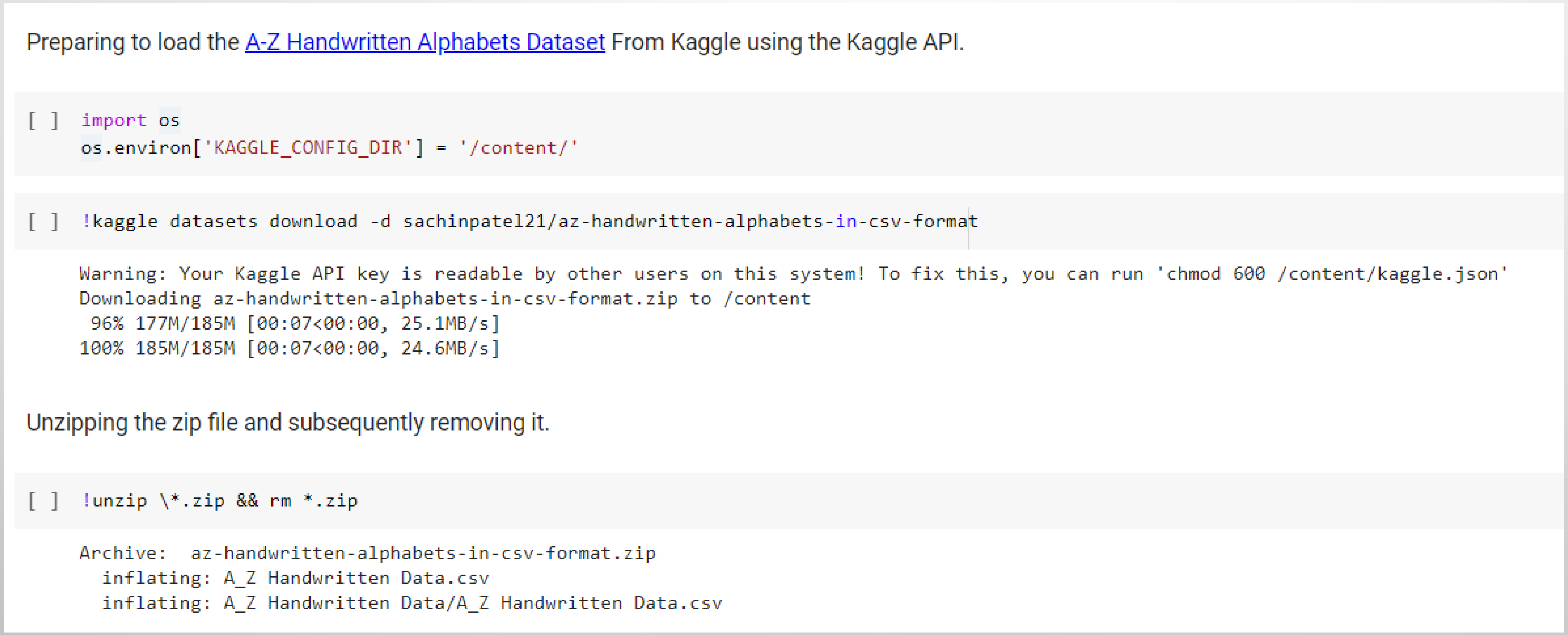
Project Description

The process of making this project is divided into 4 parts, namely:

- Getting the Data
- Pre-Processing the Data
- Creating the Model.
- Predicting an image with the Model.

Getting the Data

Preparing to the load the dataset from Kaggle using Kaggle API.



Preparing to load the A-Z Handwritten Alphabets Dataset From Kaggle using the Kaggle API.

```
[ ] import os
    os.environ['KAGGLE_CONFIG_DIR'] = '/content/'
```

```
[ ] !kaggle datasets download -d sachinpatel21/az-handwritten-alphabets-in-csv-format
```

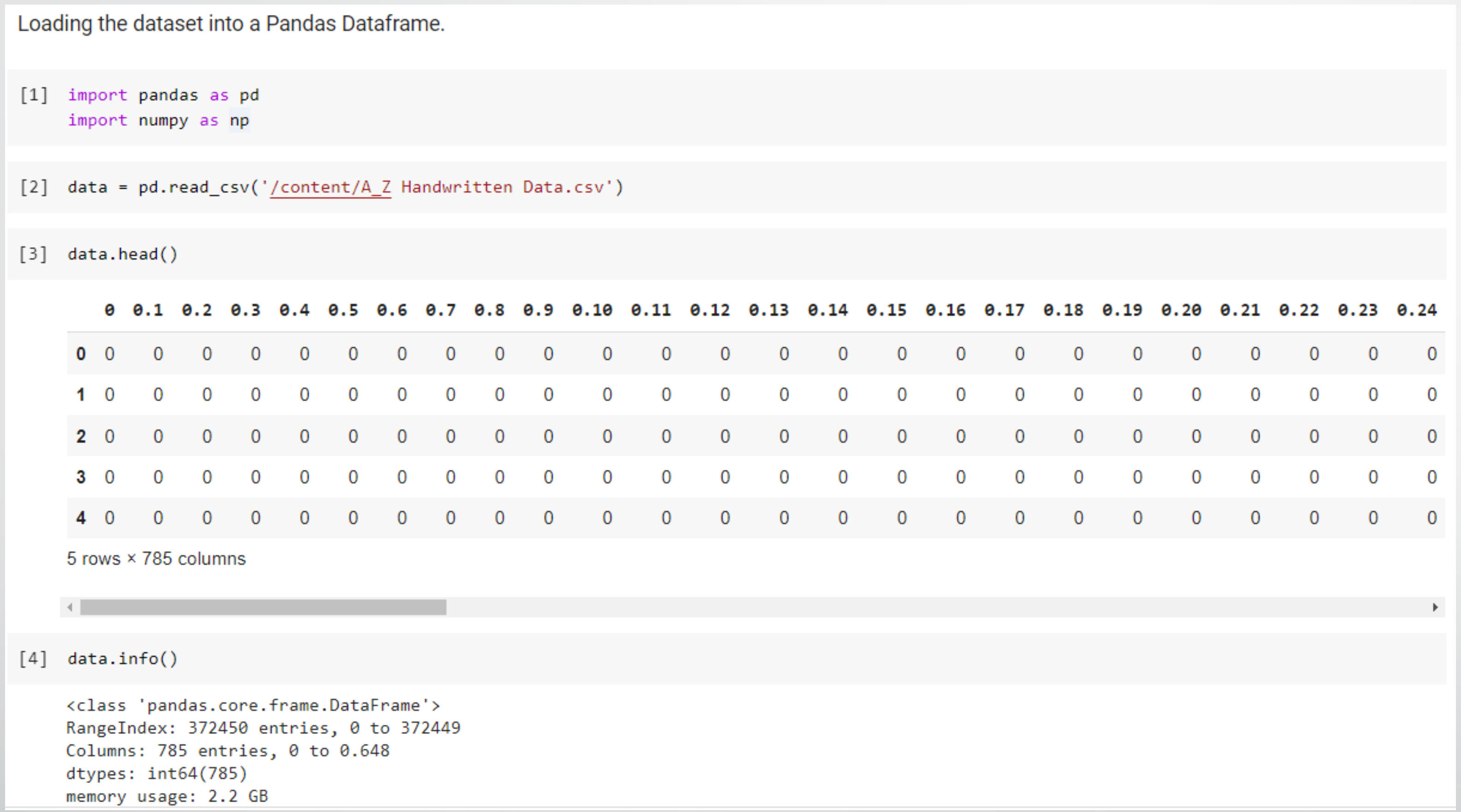
```
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /content/kaggle.json'
Downloading az-handwritten-alphabets-in-csv-format.zip to /content
 96% 177M/185M [00:07<00:00, 25.1MB/s]
100% 185M/185M [00:07<00:00, 24.6MB/s]
```

Unzipping the zip file and subsequently removing it.

```
[ ] !unzip \*.zip && rm *.zip
```

```
Archive:  az-handwritten-alphabets-in-csv-format.zip
  inflating: A_Z Handwritten Data.csv
  inflating: A_Z Handwritten Data/A_Z Handwritten Data.csv
```

Reading the CSV file and loading the dataset into a Pandas Data Frame.



Loading the dataset into a Pandas Dataframe.

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

```
[2]  data = pd.read_csv('/content/A_Z Handwritten Data.csv')
```

```
[3]  data.head()
```

	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15	0.16	0.17	0.18	0.19	0.20	0.21	0.22	0.23	0.24
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

5 rows × 785 columns

```
[4]  data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 372450 entries, 0 to 372449
Columns: 785 entries, 0 to 0.648
dtypes: int64(785)
memory usage: 2.2 GB
```

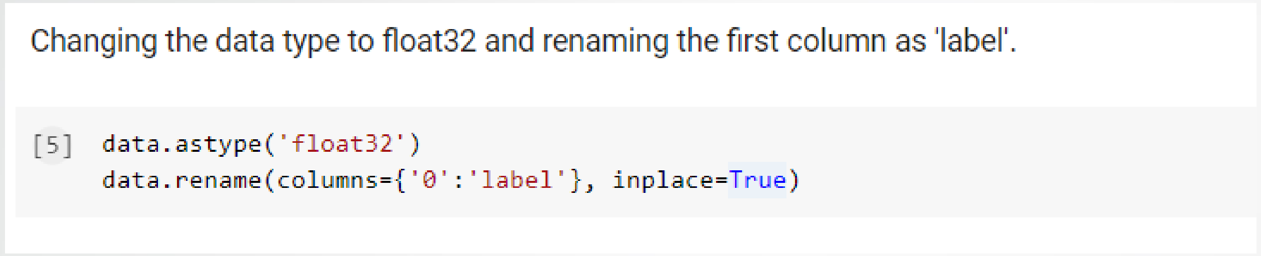
The CSV file is read into the data frame, ‘data’ which consists of 372450 entries and 785 columns.

The first column is the target column whose values range from 0-25.

E.g. For the letter ‘A’ the corresponding target column value will be 0, for the letter ‘B’ the value will be 1 and so on.

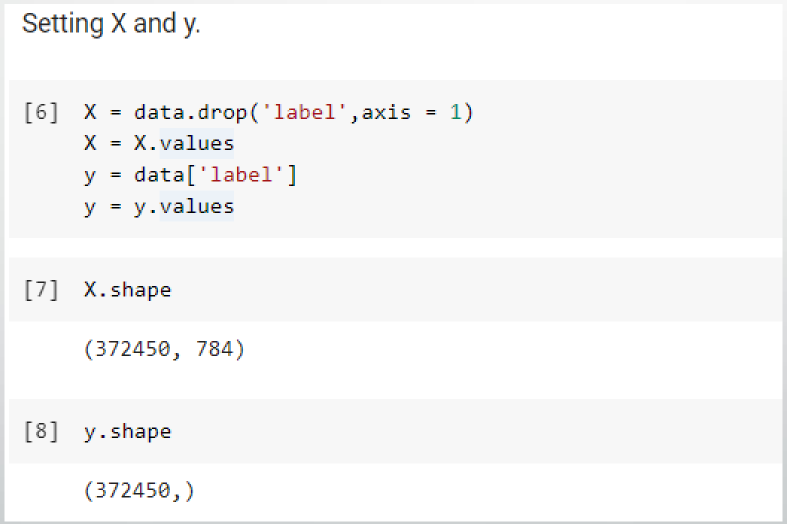
Pre-Processing the Data

Changing the data type of the data from int64 to float32 followed by changing the name of the first column to “label”.


Changing the data type to float32 and renaming the first column as 'label'.

```
[5]  data.astype('float32')
     data.rename(columns={'0':'label'}, inplace=True)
```

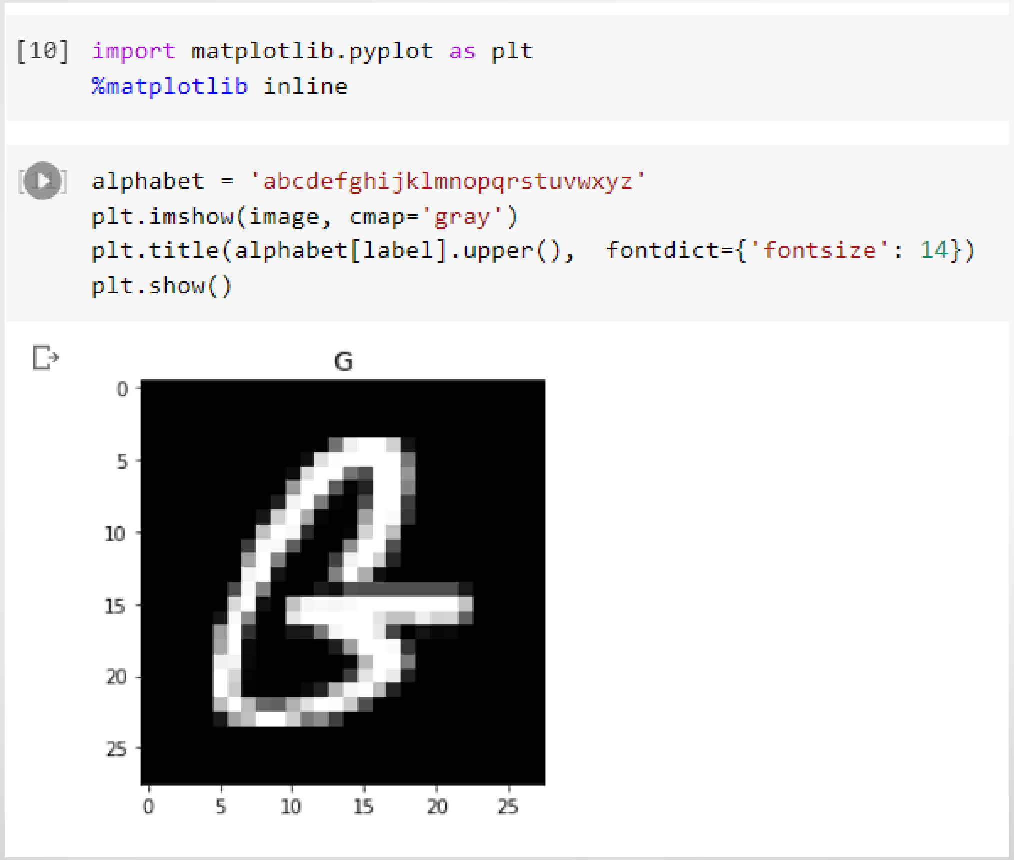
Setting the values of X and y. Here X is the "Explanatory data“ and y is the “Target”.


Setting X and y.

```
[6]  X = data.drop('label',axis = 1)
     X = X.values
     y = data['label']
     y = y.values

[7]  X.shape

     (372450, 784)

[8]  y.shape

     (372450,)
```

The data is already in the required shape, hence flattening will not be required.

Visualising an image of the dataset, say for example the image at index – 70050. For this we will be using Matplotlib.

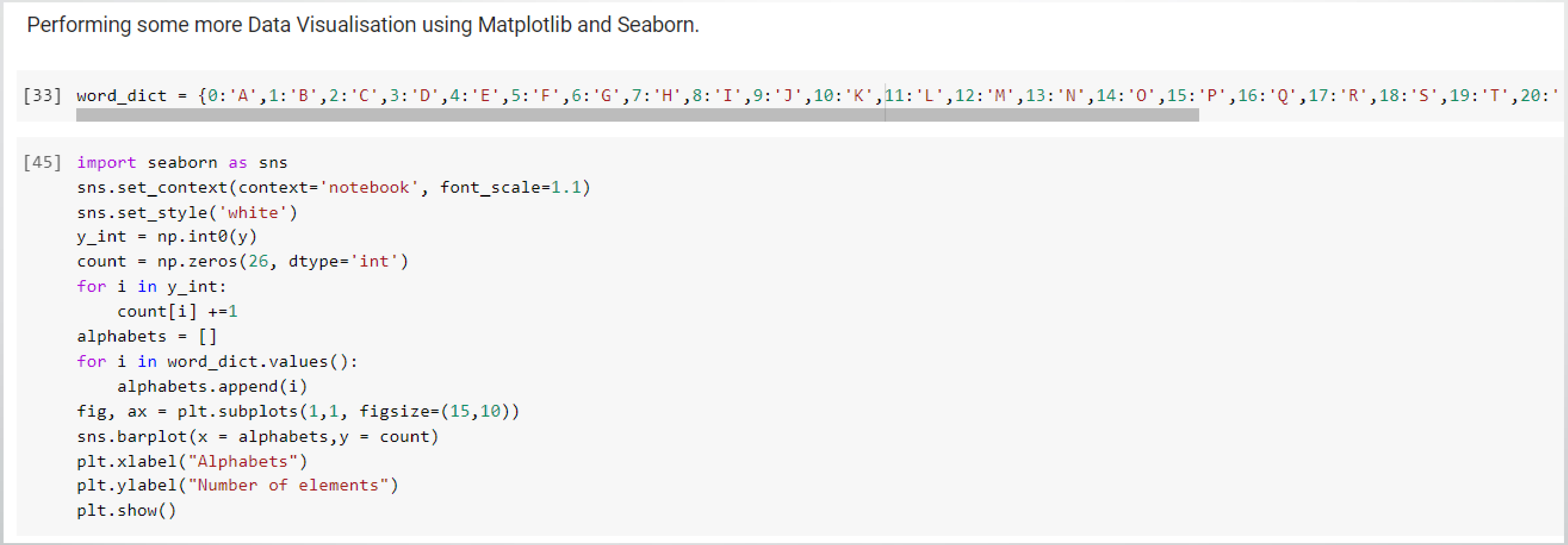


```
[10] import matplotlib.pyplot as plt
     %matplotlib inline
```

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
plt.imshow(image, cmap='gray')
plt.title(alphabet[label].upper(),  fontdict={'fontsize': 14})
plt.show()
```

This is clearly an image of the letter ‘G’ .

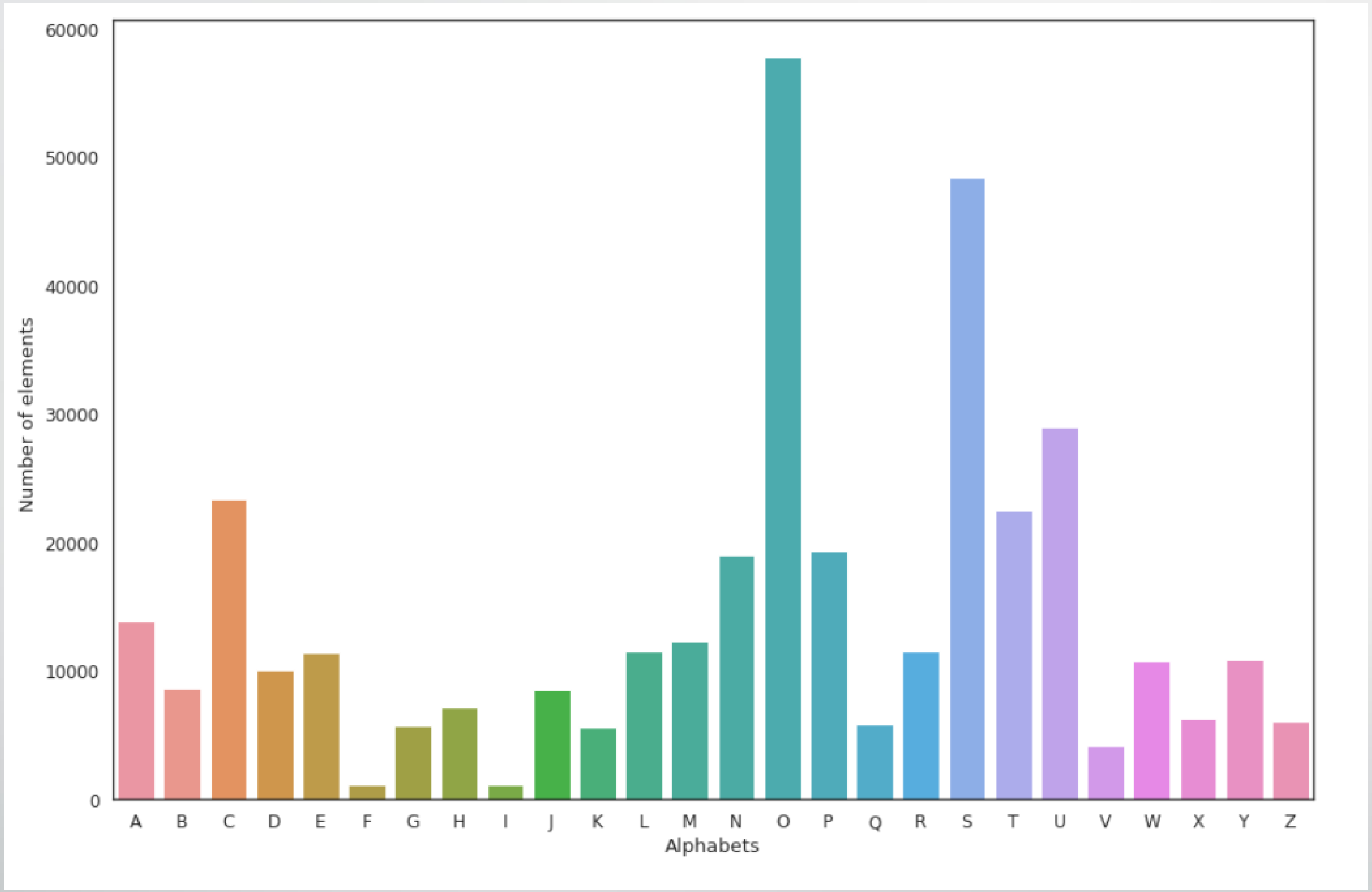
Now, performing some Data Visualisation and Analysis using Matplotlib and Seaborn.



Performing some more Data Visualisation using Matplotlib and Seaborn.

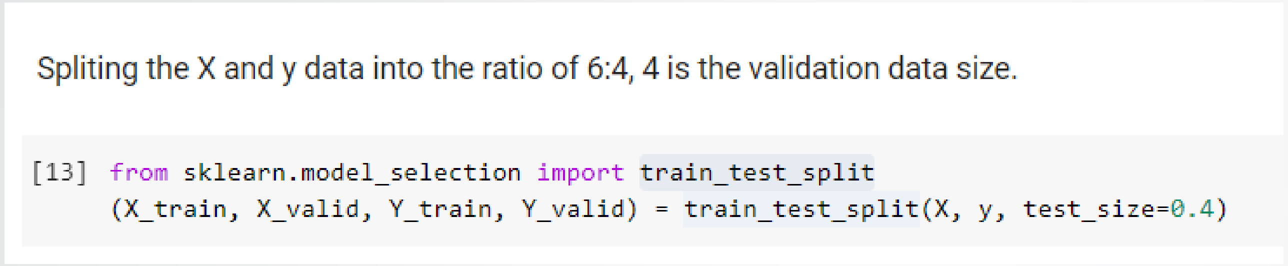
```
[33] word_dict = {0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11:'L',12:'M',13:'N',14:'O',15:'P',16:'Q',17:'R',18:'S',19:'T',20:'
```

```
[45] import seaborn as sns
     sns.set_context(context='notebook', font_scale=1.1)
     sns.set_style('white')
     y_int = np.int0(y)
     count = np.zeros(26, dtype='int')
     for i in y_int:
         count[i] +=1
     alphabets = []
     for i in word_dict.values():
         alphabets.append(i)
     fig, ax = plt.subplots(1,1, figsize=(15,10))
     sns.barplot(x = alphabets,y = count)
     plt.xlabel("Alphabets")
     plt.ylabel("Number of elements")
     plt.show()
```



It is observed that the dataset contains maximum number of images of “O”, followed by “S” and then “U”.

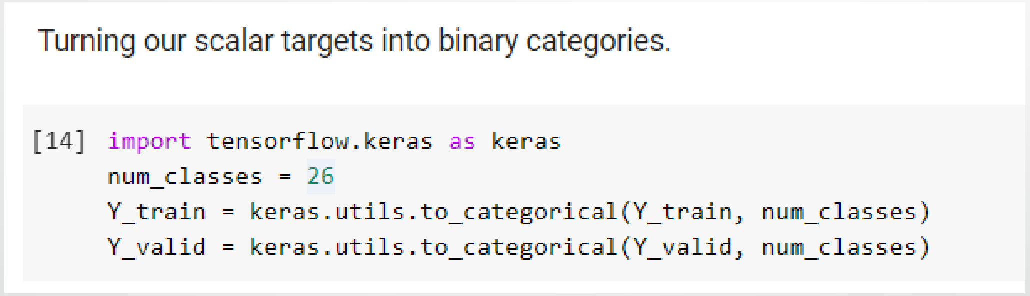
Splitting the X and y data into the training data and validation data in the ratio of 6:4. this is accomplished using the “train_test_split” function.



Spliting the X and y data into the ratio of 6:4, 4 is the validation data size.

```
[13] from sklearn.model_selection import train_test_split
     (X_train, X_valid, Y_train, Y_valid) = train_test_split(X, y, test_size=0.4)
```

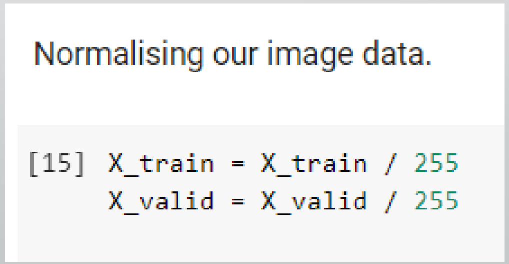
One-Hot-Encoding of the target.



Turning our scalar targets into binary categories.

```
[14] import tensorflow.keras as keras
     num_classes = 26
     Y_train = keras.utils.to_categorical(Y_train, num_classes)
     Y_valid = keras.utils.to_categorical(Y_valid, num_classes)
```

The pixel data ranges from 0 to 255, where 0 is black and 255 is white. Normalizing the data to the range 0 - 1.

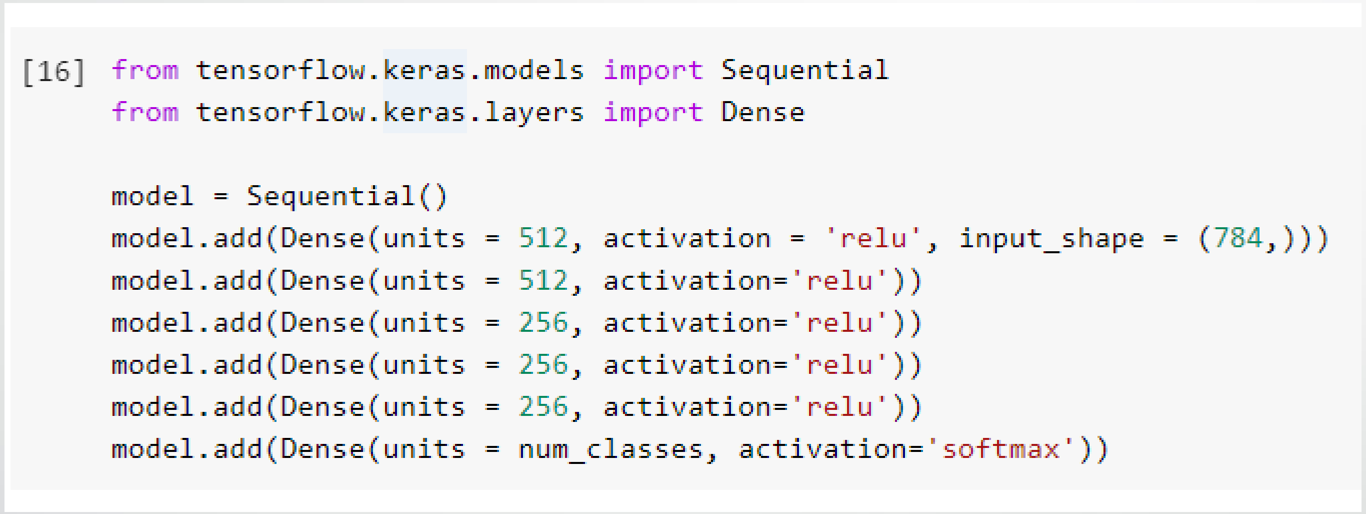


Normalising our image data.

```
[15] X_train = X_train / 255
     X_valid = X_valid / 255
```

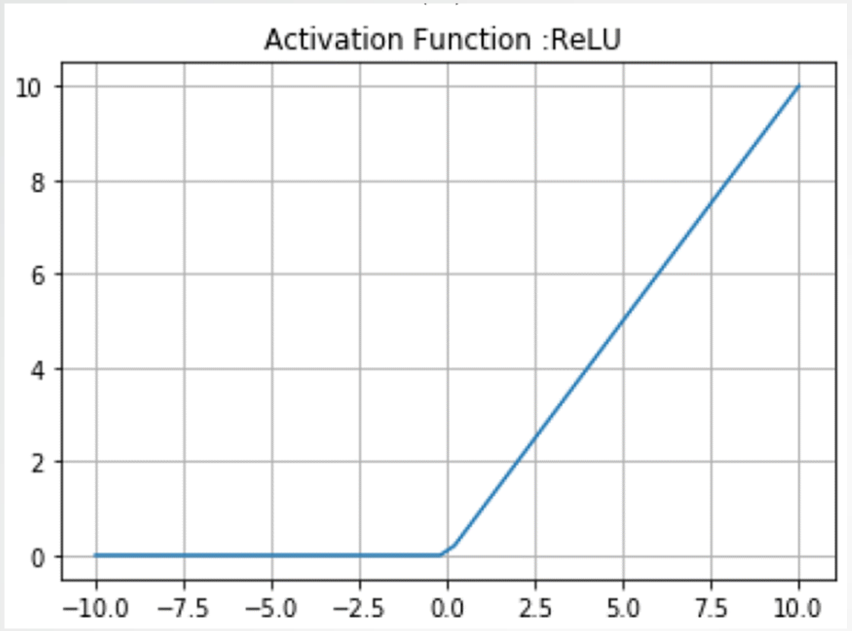
Creating the Model

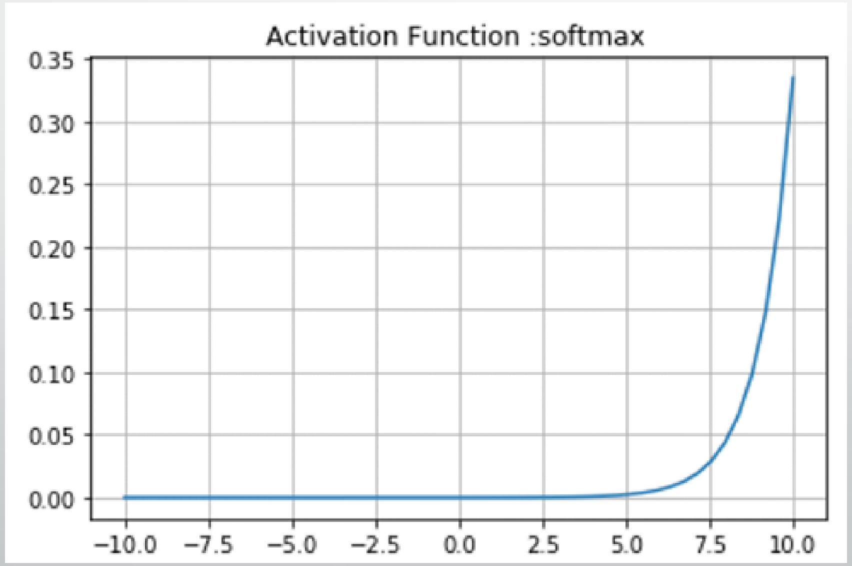
Creating a MLP Model comprising of an input layer, four hidden layers and an output layer.



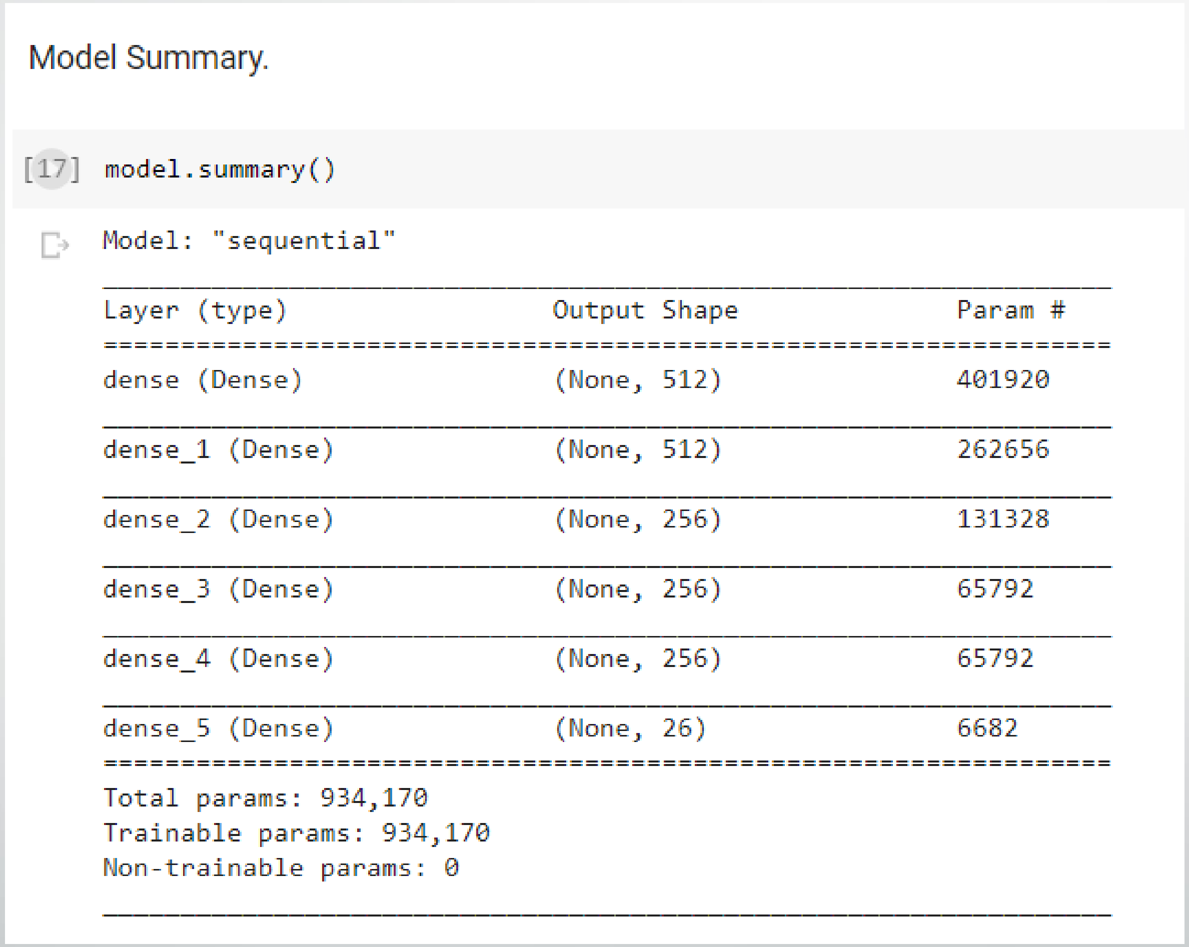
```
[16] from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense

     model = Sequential()
     model.add(Dense(units = 512, activation = 'relu', input_shape = (784,)))
     model.add(Dense(units = 512, activation='relu'))
     model.add(Dense(units = 256, activation='relu'))
     model.add(Dense(units = 256, activation='relu'))
     model.add(Dense(units = 256, activation='relu'))
     model.add(Dense(units = num_classes, activation='softmax'))
```

The input and the hidden layers have “ReLU” as the activation function and the output layer has “softmax” as the activation function.


Activation Function :ReLU
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Activation Function :softmax
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

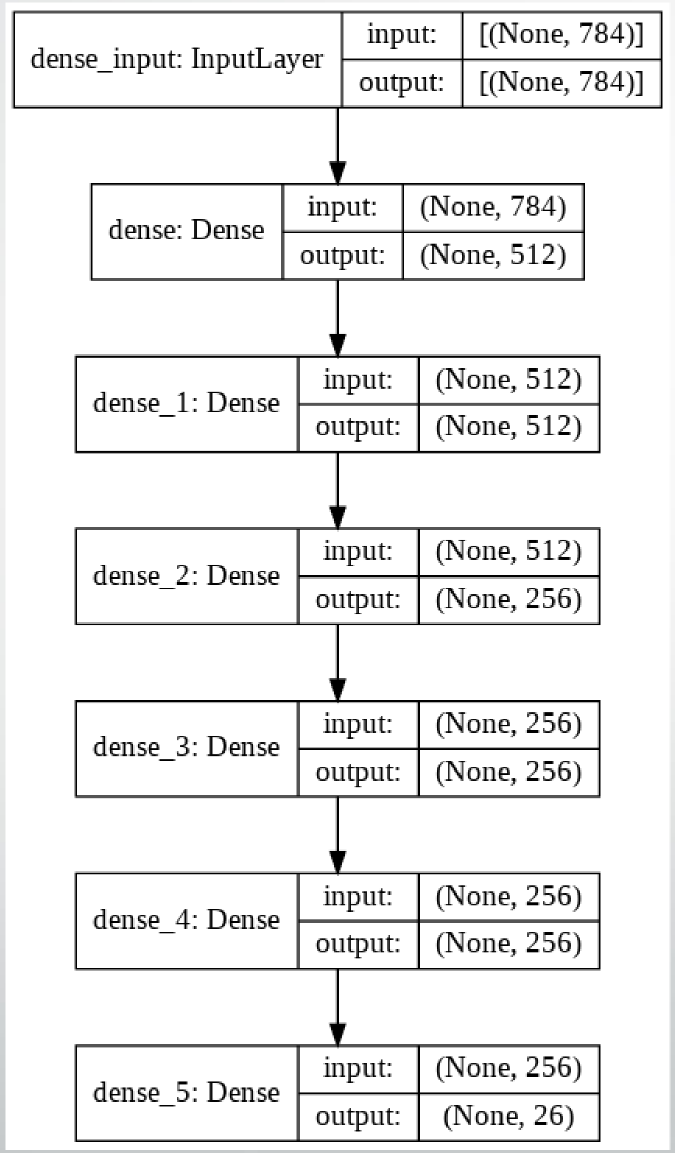
The Model Summary:


Model Summary.

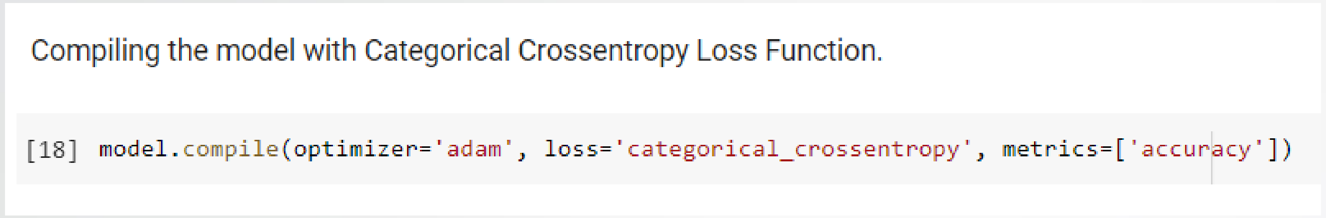
```
[17] model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 512)               401920
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_3 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_4 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_5 (Dense)              (None, 26)                6682
=================================================================
Total params: 934,170
Trainable params: 934,170
Non-trainable params: 0
_________________________________________________________________
```

The Pictorial Representation of the Model:



Compiling the model with Adam as the optimizing function. The Loss function used is Categorical Crossentropy.


Compiling the model with Categorical Crossentropy Loss Function.

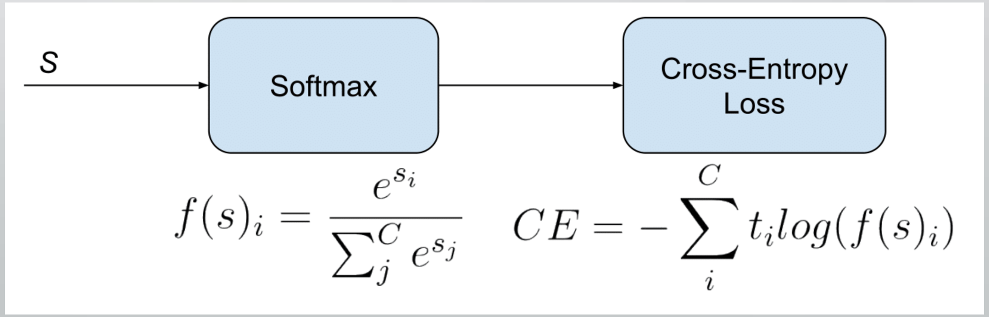
```
[18] model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Adam Optimizer

Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the 'gradient descent with momentum' algorithm and the 'RMSP' algorithm.

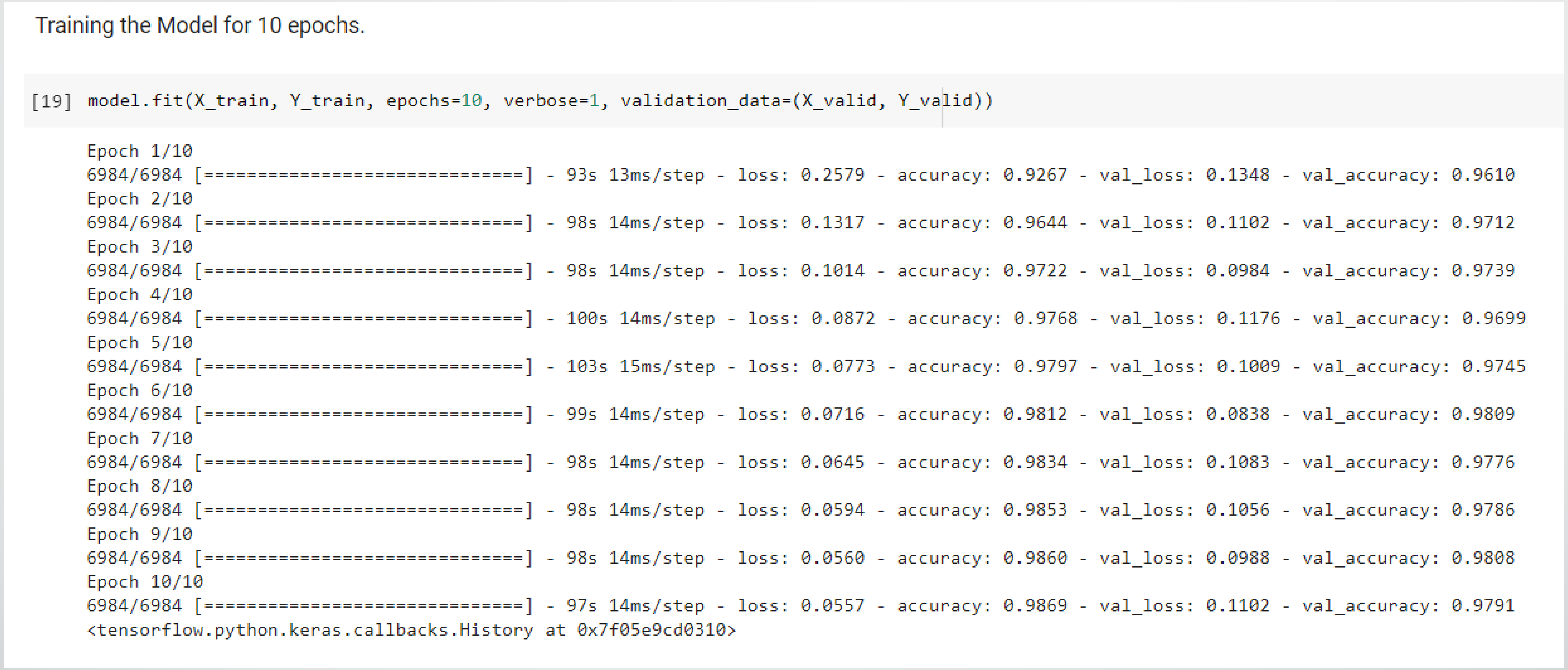
Categorical Crossentropy

Also called Softmax Loss. It is a Softmax activation plus a Cross-Entropy loss. It is used for multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

Now, training the model for 10 epochs.


Training the Model for 10 epochs.

```
[19] model.fit(X_train, Y_train, epochs=10, verbose=1, validation_data=(X_valid, Y_valid))
```

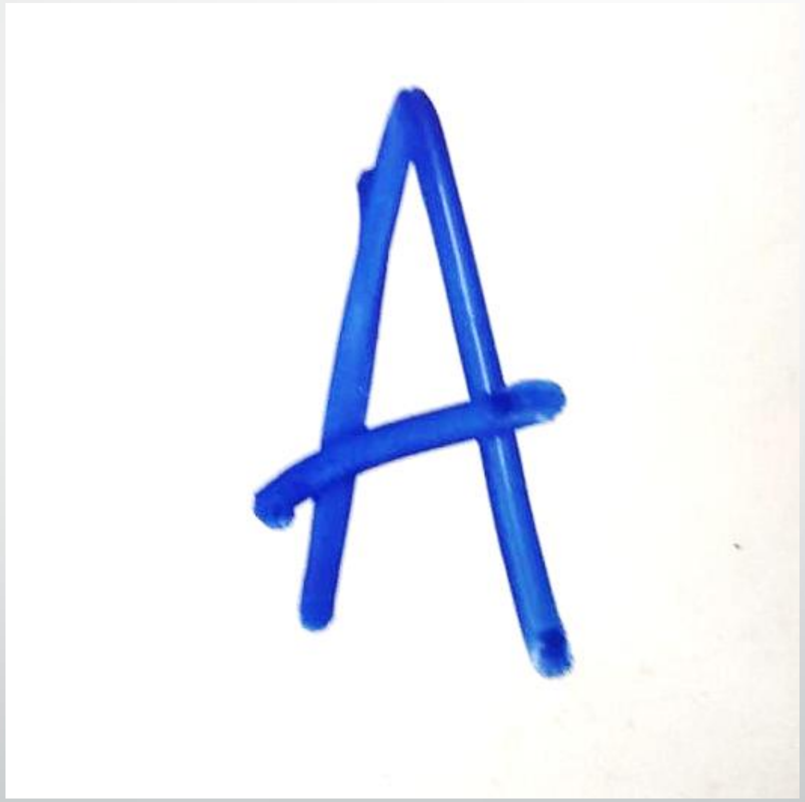
```
Epoch 1/10
6984/6984 [==============================] - 93s 13ms/step - loss: 0.2579 - accuracy: 0.9267 - val_loss: 0.1348 - val_accuracy: 0.9610
Epoch 2/10
6984/6984 [==============================] - 98s 14ms/step - loss: 0.1317 - accuracy: 0.9644 - val_loss: 0.1102 - val_accuracy: 0.9712
Epoch 3/10
6984/6984 [==============================] - 98s 14ms/step - loss: 0.1014 - accuracy: 0.9722 - val_loss: 0.0984 - val_accuracy: 0.9739
Epoch 4/10
6984/6984 [==============================] - 100s 14ms/step - loss: 0.0872 - accuracy: 0.9768 - val_loss: 0.1176 - val_accuracy: 0.9699
Epoch 5/10
6984/6984 [==============================] - 103s 15ms/step - loss: 0.0773 - accuracy: 0.9797 - val_loss: 0.1009 - val_accuracy: 0.9745
Epoch 6/10
6984/6984 [==============================] - 99s 14ms/step - loss: 0.0716 - accuracy: 0.9812 - val_loss: 0.0838 - val_accuracy: 0.9809
Epoch 7/10
6984/6984 [==============================] - 98s 14ms/step - loss: 0.0645 - accuracy: 0.9834 - val_loss: 0.1083 - val_accuracy: 0.9776
Epoch 8/10
6984/6984 [==============================] - 98s 14ms/step - loss: 0.0594 - accuracy: 0.9853 - val_loss: 0.1056 - val_accuracy: 0.9786
Epoch 9/10
6984/6984 [==============================] - 98s 14ms/step - loss: 0.0560 - accuracy: 0.9860 - val_loss: 0.0988 - val_accuracy: 0.9808
Epoch 10/10
6984/6984 [==============================] - 97s 14ms/step - loss: 0.0557 - accuracy: 0.9869 - val_loss: 0.1102 - val_accuracy: 0.9791
<tensorflow.python.keras.callbacks.History at 0x7f05e9cd0310>
```

After training for 10 epochs,

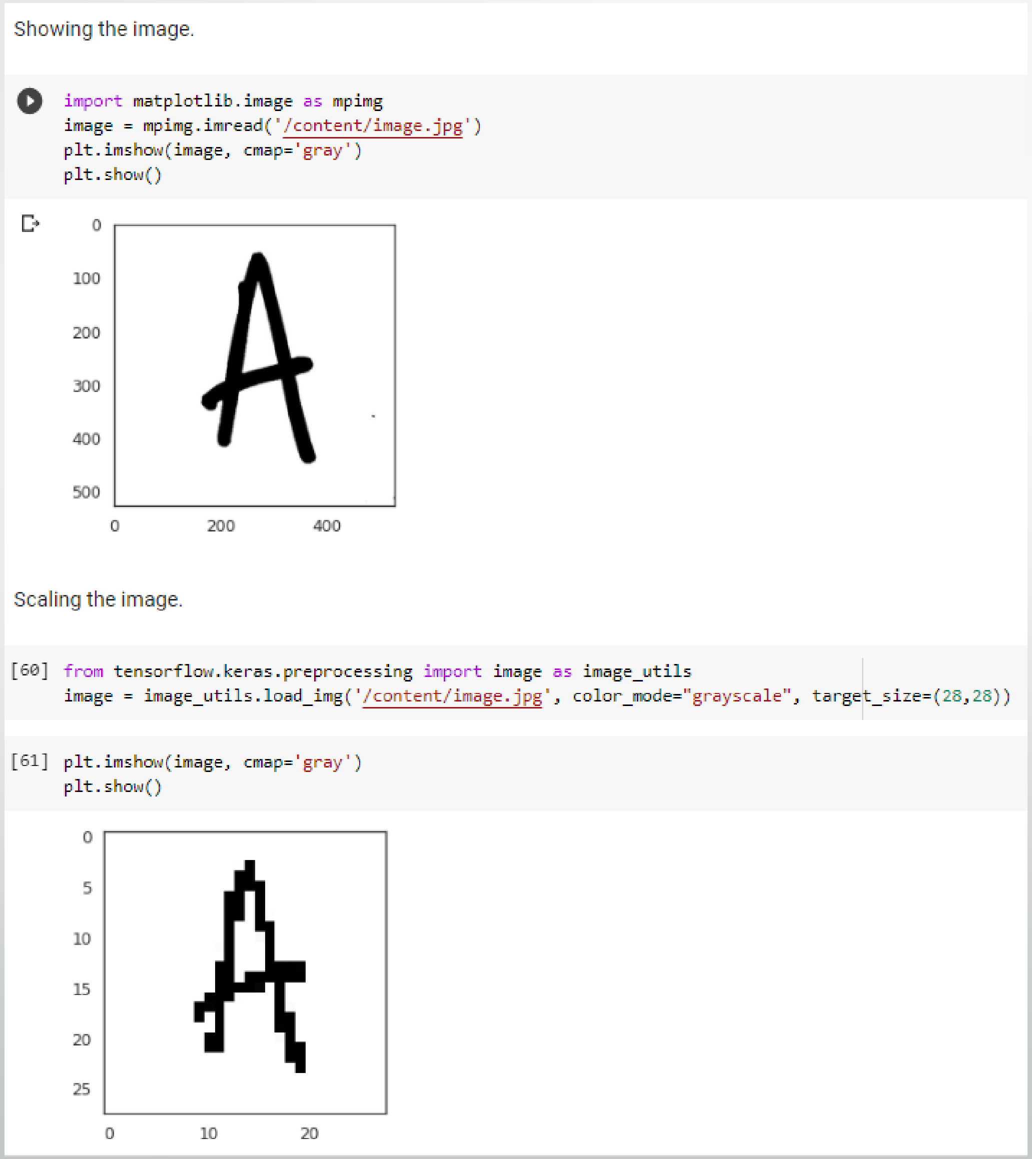
- Training Accuracy: 0.9869
- Training Loss: 0.0557
- Validation Accuracy: 0.9791
- Validation Loss: 0.1102

Predicting an image with the Model

Predicting the following image (taken with my phone camera) using the model:

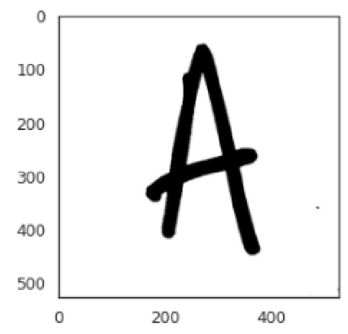


The images in our dataset were 28x28 pixels and grayscale. The same size and grayscale images need to be passed into the model for prediction. This is achieved using Keras' built-in utility that works well.



Showing the image.

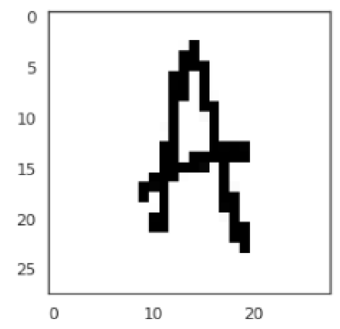
```
import matplotlib.image as mpimg
image = mpimg.imread('/content/image.jpg')
plt.imshow(image, cmap='gray')
plt.show()
```



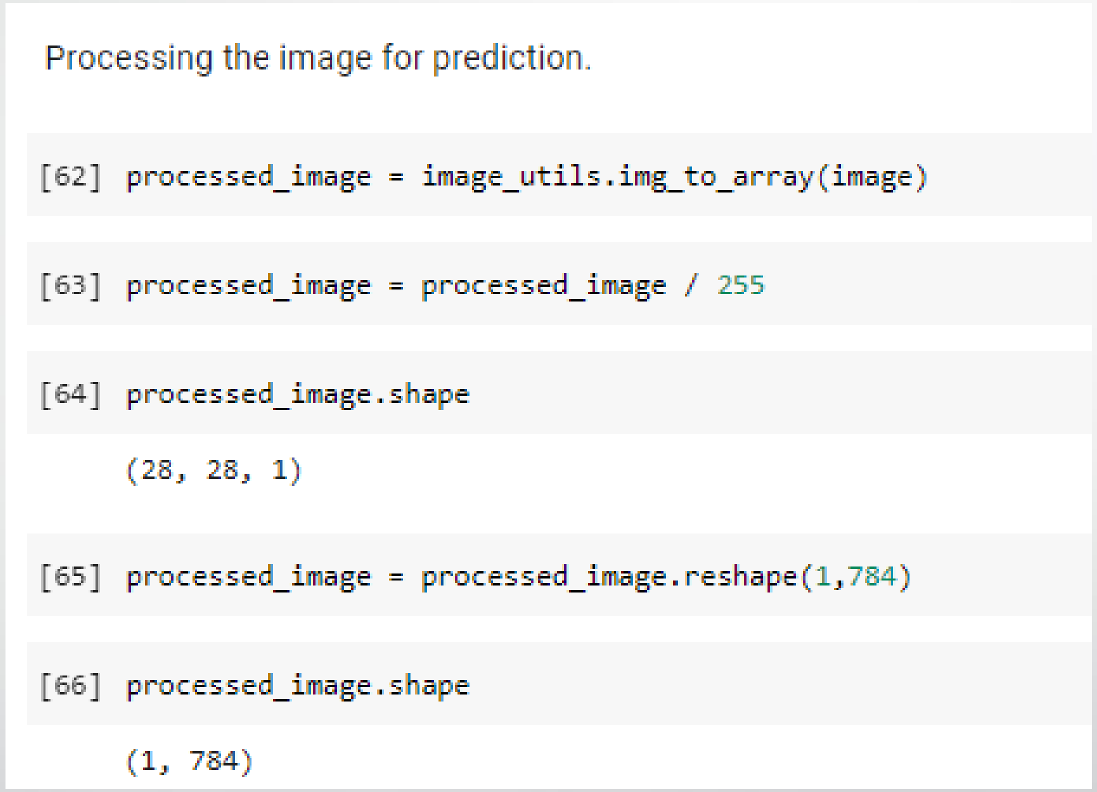
Scaling the image.

```
[60] from tensorflow.keras.preprocessing import image as image_utils
     image = image_utils.load_img('/content/image.jpg', color_mode="grayscale", target_size=(28,28))
```

```
[61] plt.imshow(image, cmap='gray')
     plt.show()
```



Converting the image into a more rudimentary format using Keras utility called “image_to_array”, followed by reshaping the image to the correct shape.



Processing the image for prediction.

```
[62] processed_image = image_utils.img_to_array(image)
```

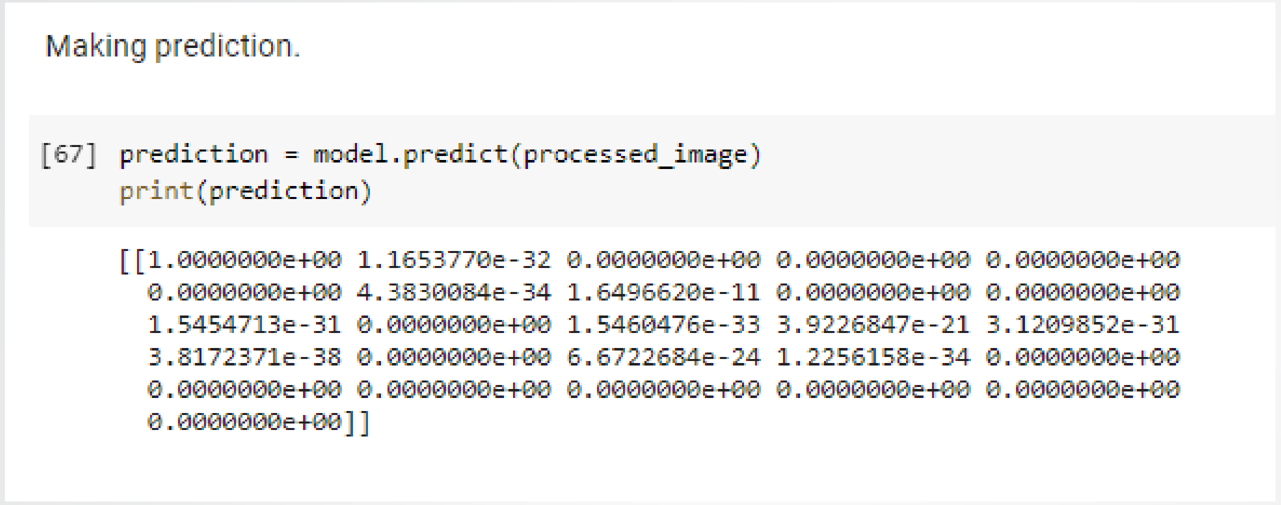
```
[63] processed_image = processed_image / 255
```

```
[64] processed_image.shape

     (28, 28, 1)
```

```
[65] processed_image = processed_image.reshape(1,784)
```

```
[66] processed_image.shape

     (1, 784)
```

Finally, making the prediction.



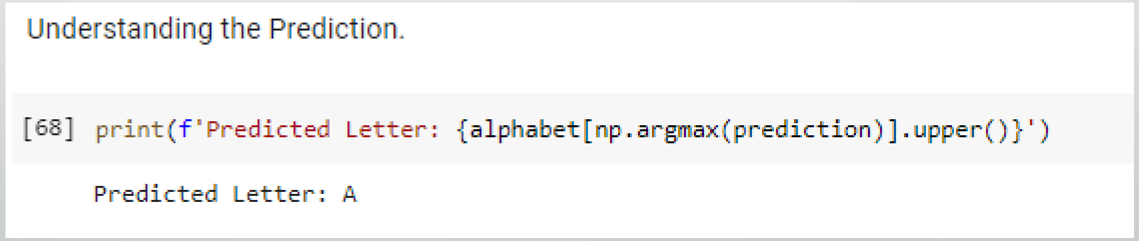
Making prediction.

```
[67] prediction = model.predict(processed_image)
     print(prediction)
```

```
[[1.0000000e+00 1.1653770e-32 0.0000000e+00 0.0000000e+00 0.0000000e+00
  0.0000000e+00 4.3830084e-34 1.6496620e-11 0.0000000e+00 0.0000000e+00
  1.5454713e-31 0.0000000e+00 1.5460476e-33 3.9226847e-21 3.1209852e-31
  3.8172371e-38 0.0000000e+00 6.6722684e-24 1.2256158e-34 0.0000000e+00
  0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00 0.0000000e+00
  0.0000000e+00]]
```

The predictions are in the format of a 26 length array. Each element of the array is a probability between 0 and 1, representing the probability for each category. The element with the highest probability is the predicted category. Using the np.argmax() function to find the element with the highest probability.

The value from the np.argmax() function is used as the index for the string, alphabet = 'abcdefghijklmnopqrstuvwxyz', to determine the correct alphabet.



Understanding the Prediction.

```
[68] print(f'Predicted Letter: {alphabet[np.argmax(prediction)].upper()}')
```

```
Predicted Letter: A
```

The predicted letter is ‘A’ which is the correct answer.

Conclusion

We have successfully developed Handwritten Character Recognition (Text Recognition) MLP model with Python, TensorFlow, and Machine Learning libraries.

Handwritten characters have been recognized with more than 98% accuracy. This can be also further extended to identify the handwritten characters of other languages as well.

References

- https://data-flair.training/blogs/handwritten-character-recognition-neural-network/
- https://www.kaggle.com/nohrud/99-9-accuracy-on-alphabet-recognition-by-eda
- https://gombru.github.io/2018/05/23/cross_entropy_loss/
- https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy
- https://www.geeksforgeeks.org/intuition-of-adam-optimizer/