Handwritten Character Recognition

(MLP + Crossentropy)

Submitted 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.

The image of the written text may be sensed "off line" from a piece of paper by optical scanning (optical character recognition) or intelligent word recognition. **Optical character recognition (OCR)** is the electronic or mechanical conversion of images of typed, handwritten or printed text into machine-encoded text, whether from a scanned document, a photo of a document, a scene-photo (for example the text on signs and billboards in a landscape photo) or from subtitle text superimposed on an image (for example: from a television broadcast).

Widely used as a form of data entry from printed paper data records – whether passport documents, invoices, bank statements, computerized receipts, business cards, mail, printouts of static-data, or any suitable documentation – it is a common method of digitizing printed texts so that they can be electronically edited, searched, stored more compactly, displayed on-line, and used in machine processes such as cognitive computing, machine translation, (extracted) text-to-speech, key data and text mining. OCR is a field of research in pattern recognition, artificial intelligence and computer vision.

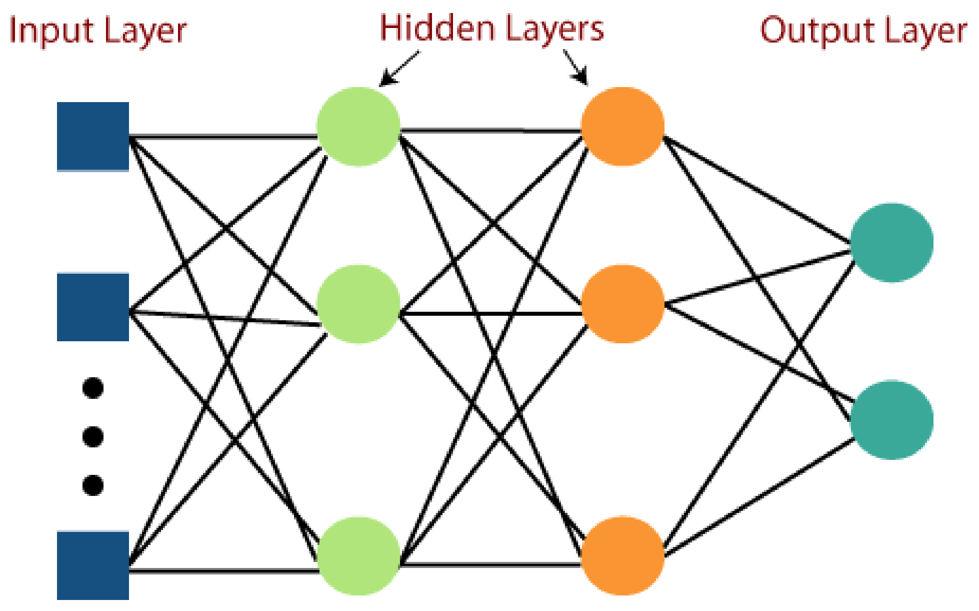
Objective of the project

The project aims at recognizing handwritten characters, i.e., English alphabets from A-Z, 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.

Multi-Layer Perceptron

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.



Activation function

If a multilayer perceptron has a linear activation function in all neurons, that is, a linear function that maps the weighted inputs to the output of each neuron, then linear algebra shows that any number of layers can be reduced to a two-layer input-output model. In MLPs some neurons use a nonlinear activation function that was developed to model the frequency of action potentials, or firing, of biological neurons.

The sigmoid function has been historically used as a common activation function; but in recent developments of deep learning the rectifier linear unit (ReLU) is more frequently used as one of the possible ways to overcome the numerical problems related to the sigmoid function.

Layers

The MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of nonlinearly-activating nodes. Since MLPs are fully connected, each node in one layer connects with a certain weight to every node in the following layer.

Learning

Learning occurs in the perceptron by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning, and is carried out through backpropagation, a generalization of the least mean squares algorithm in the linear perceptron.

Project Prerequisites

The following are the prerequisites for this project:

1. Python (3.7)
2. IDE – Jupyter Notebook (Google Colab)
3. Frameworks used:
    a. NumPy (1.19.5)
    b. Pandas (1.1.5)
    c. Keras (2.5.0)
    d. TensorFlow (Keras uses TensorFlow in backend and for some image pre-processing) (2.5.0)
    e. Matplotlib (3.2.2)
    f. 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.

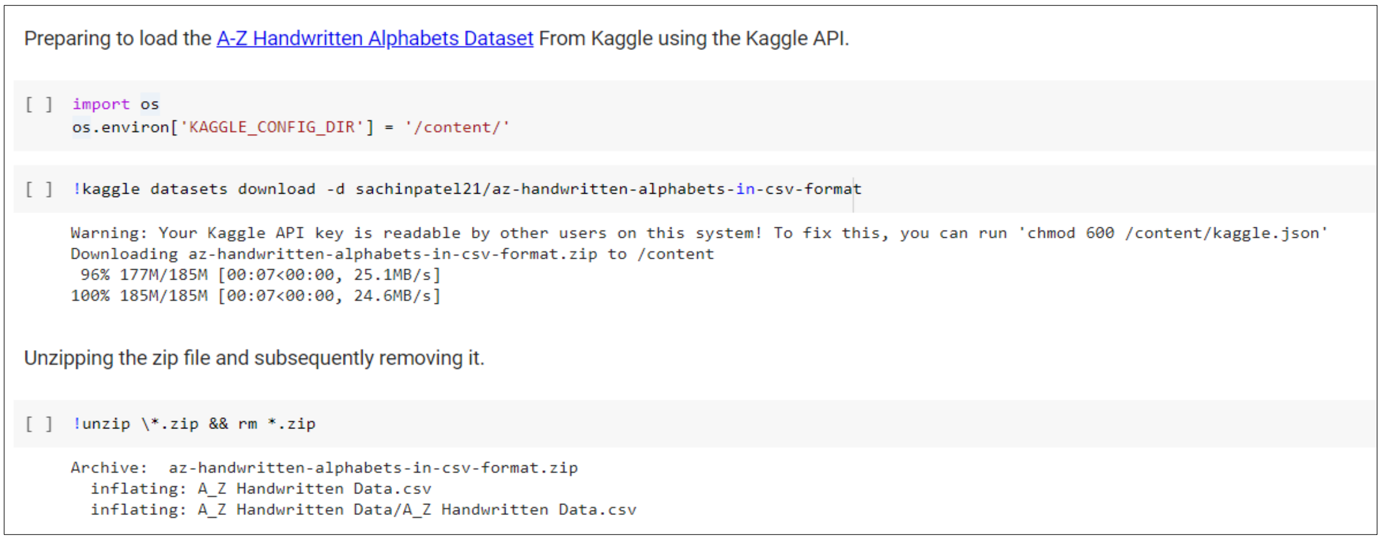
Implementing the Project

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

1. Getting the Data
2. Pre-Processing the Data
3. Creating the Model.
4. Predicting an image with the Model

Getting the Data

The data is directly imported from Kaggle using the 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
```

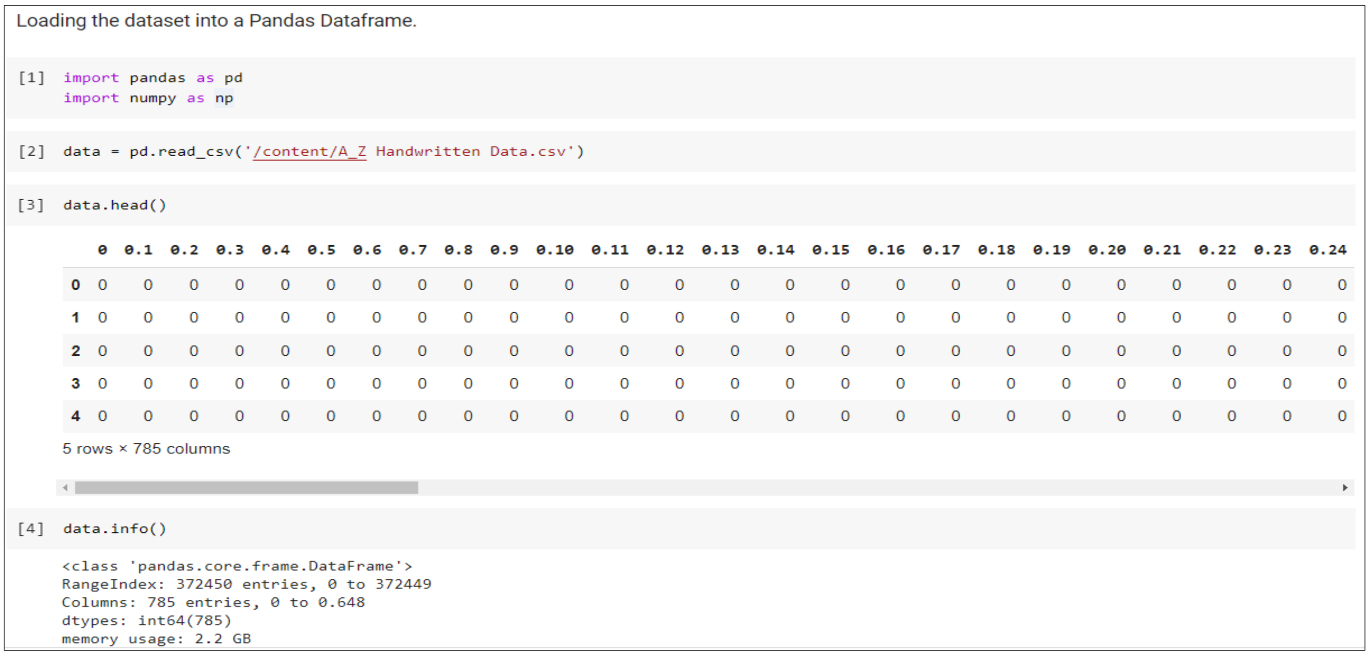
```
Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /content/kaggle.json'
Downloading az-handwritten-alphabets-in-csv-format.zip to /content
 96% 177M/185M [00:07<00:00, 25.1MB/s]
100% 185M/185M [00:07<00:00, 24.6MB/s]
```

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
```

After which the dataset CSV file is loaded the dataset into a Pandas data frame, named "data".



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()
```

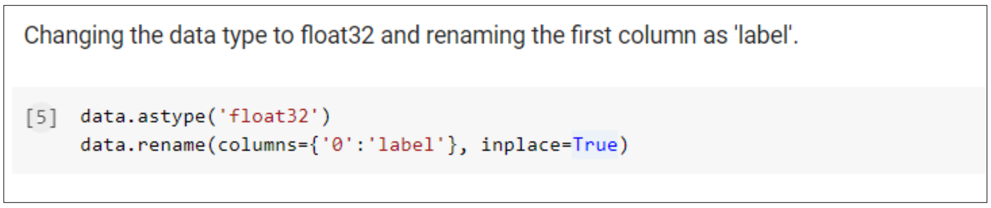
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 372450 entries, 0 to 372449
Columns: 785 entries, 0 to 0.648
dtypes: int64(785)
memory usage: 2.2 GB
```

The data frame, "data" 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

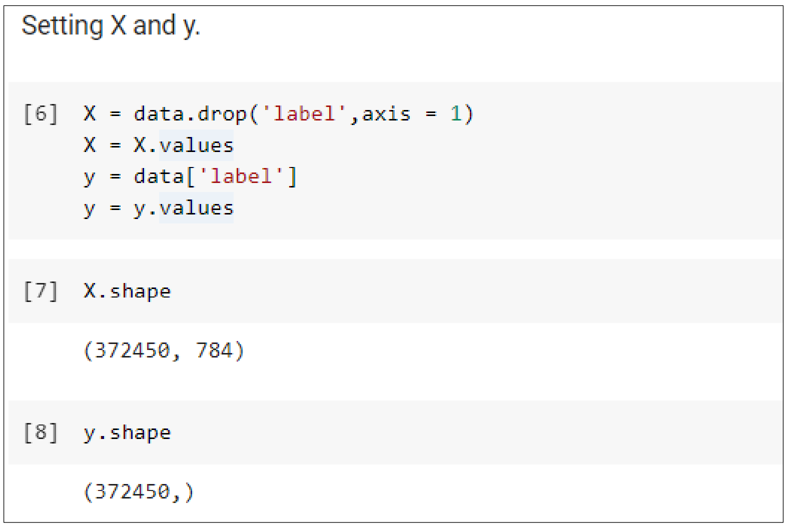
The data type of the entries is int64 and must be changed to float32. The name of the first column is also changed to “label”.



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

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

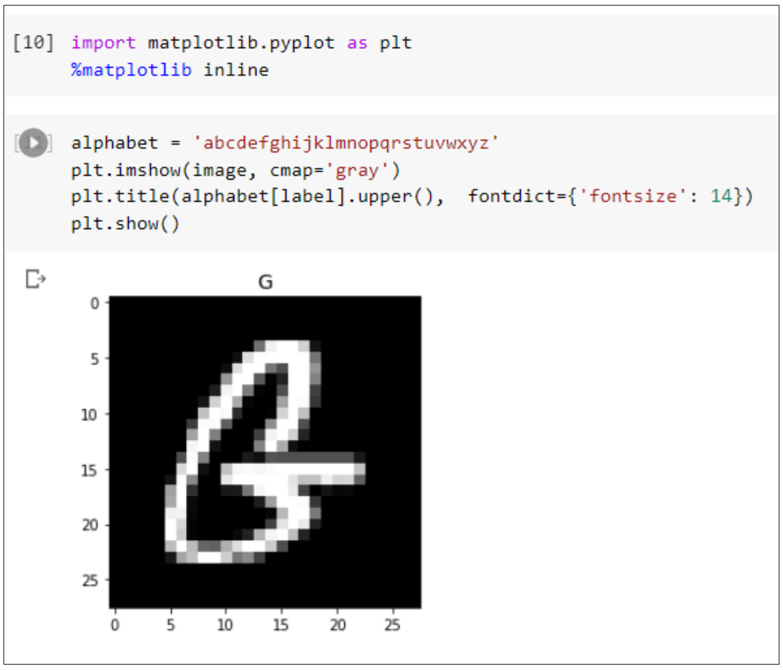
“X” is set as the Explanatory data and “y“, is the Target. The data is already in the required shape, hence flattening will not be required.



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,)
```

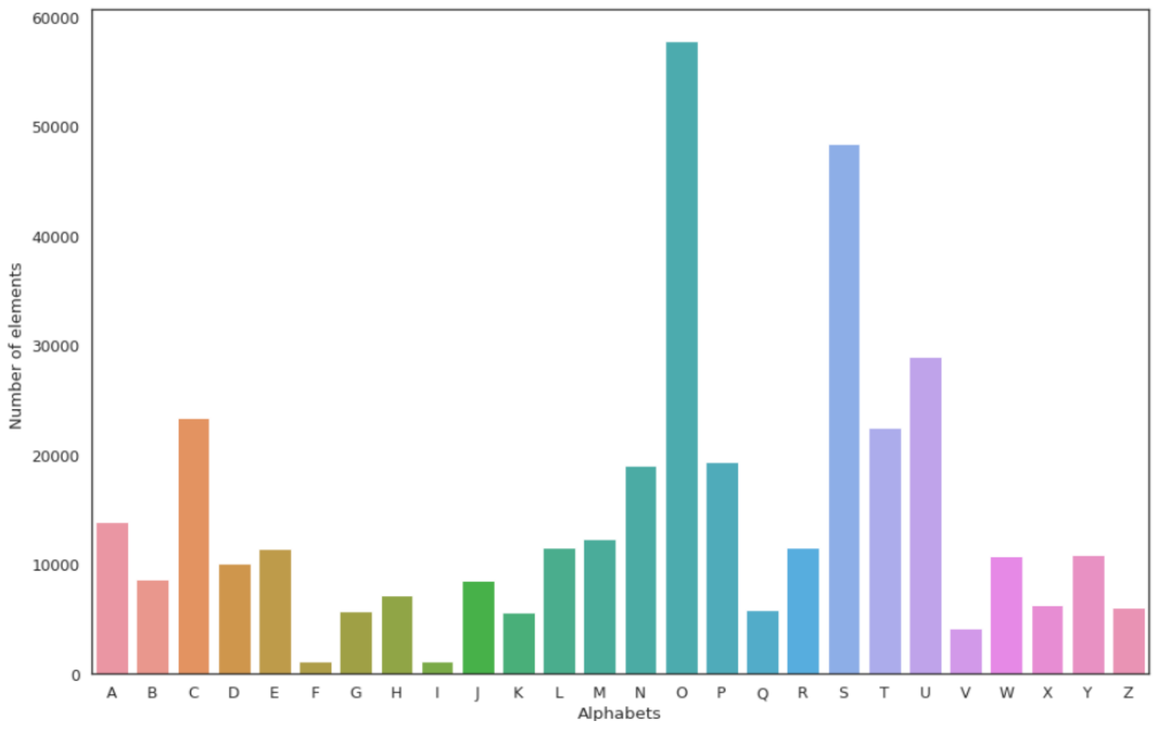
Visualising an image of the dataset, say for example the image at index – 70050. For this purpose, Matplotlib is used. This is clearly an image of the letter 'G'.



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

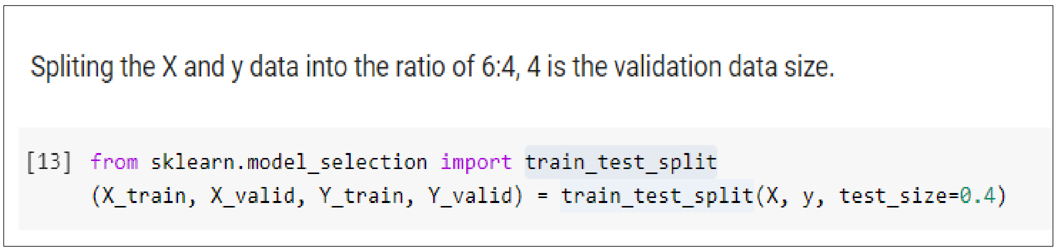
```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
plt.imshow(image, cmap='gray')
plt.title(alphabet[label].upper(),  fontdict={'fontsize': 14})
plt.show()
```

Visualising the distribution of images of different letters int e dataset using Matplotlib and Seaborn.



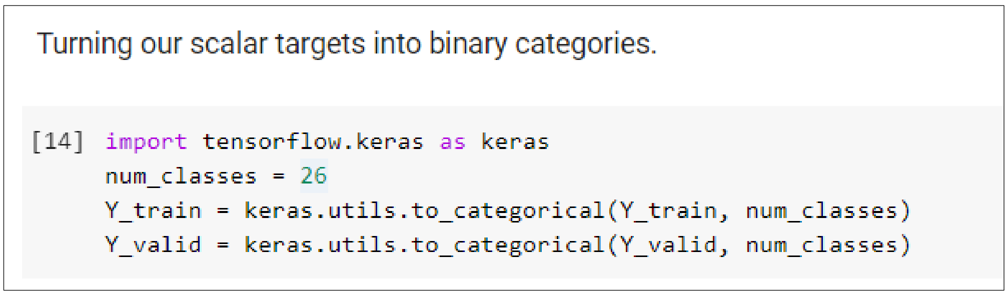
It is observed that the dataset contains maximum number of images of "O", followed by "S" and then "U".

The X and y data is split 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.

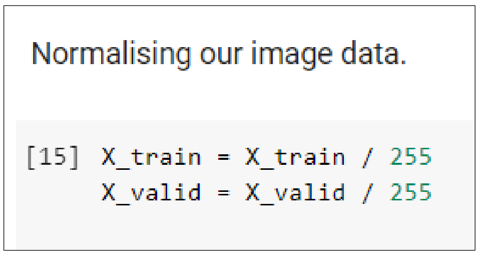
```
[13] from sklearn.model_selection import train_test_split
     (X_train, X_valid, Y_train, Y_valid) = train_test_split(X, y, test_size=0.4)
```

The scaler targets are turned into binary categories comprising of 26 classes using **one-hot-encoding** which is a frequently used method to deal with categorical data. Since there is no quantitative relationship between individual target values, one-hot encoding is applied to the target values, in order to improve the performance of the algorithm. Using ordinal encoding can potentially create a fictional ordinal relationship in the data, thus leading to decrease in efficiency.


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 X data is finally normalised to the range 0-1. This is easily achieved by dividing the X training and X validation data by the value 255 which is the maximum pixel data value.


Normalising our image data.

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

Creating the Model

The MLP Model comprises of six Dense layers: an input layer, four hidden layers and an output layer. The input and the hidden layers have "**ReLU**" as the activation function which has been found to enable better training of deeper networks, compared to the widely used activation functions prior to 2011, e.g., the logistic sigmoid and the hyperbolic tangent.

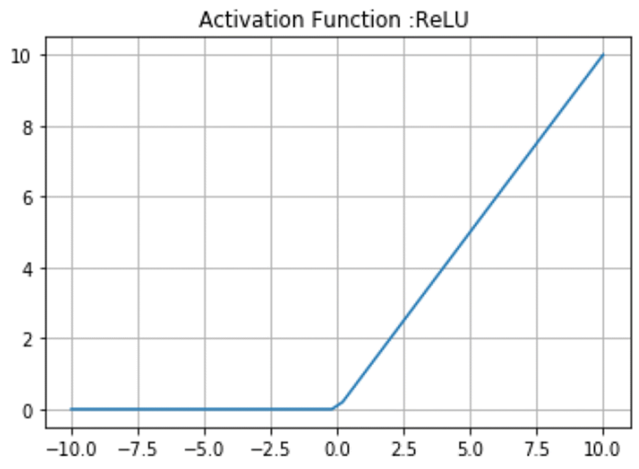
The ReLU activation function is an activation function defined as the positive part of its argument

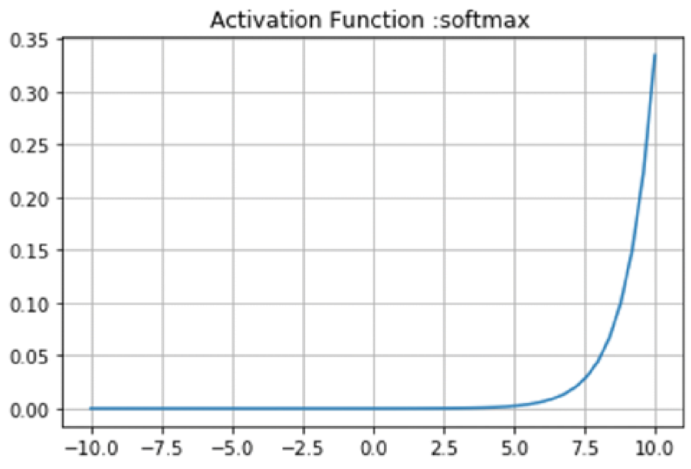
$$f(x) = x^+ = \max(0, x)$$

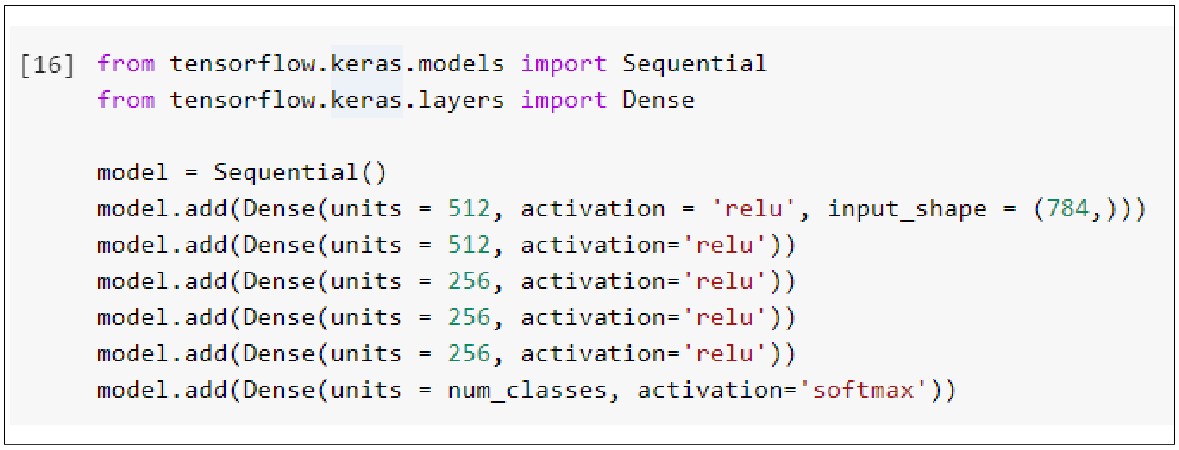
The output layer has "**softmax**" as the activation function which is a generalization of the logistic function to multiple dimensions. It is used in multinomial logistic regression and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over the predicted output classes.

The standard (unit) softmax function $\sigma : \mathbb{R}^K \to [0,1]^K$ is defined by the formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K.$$

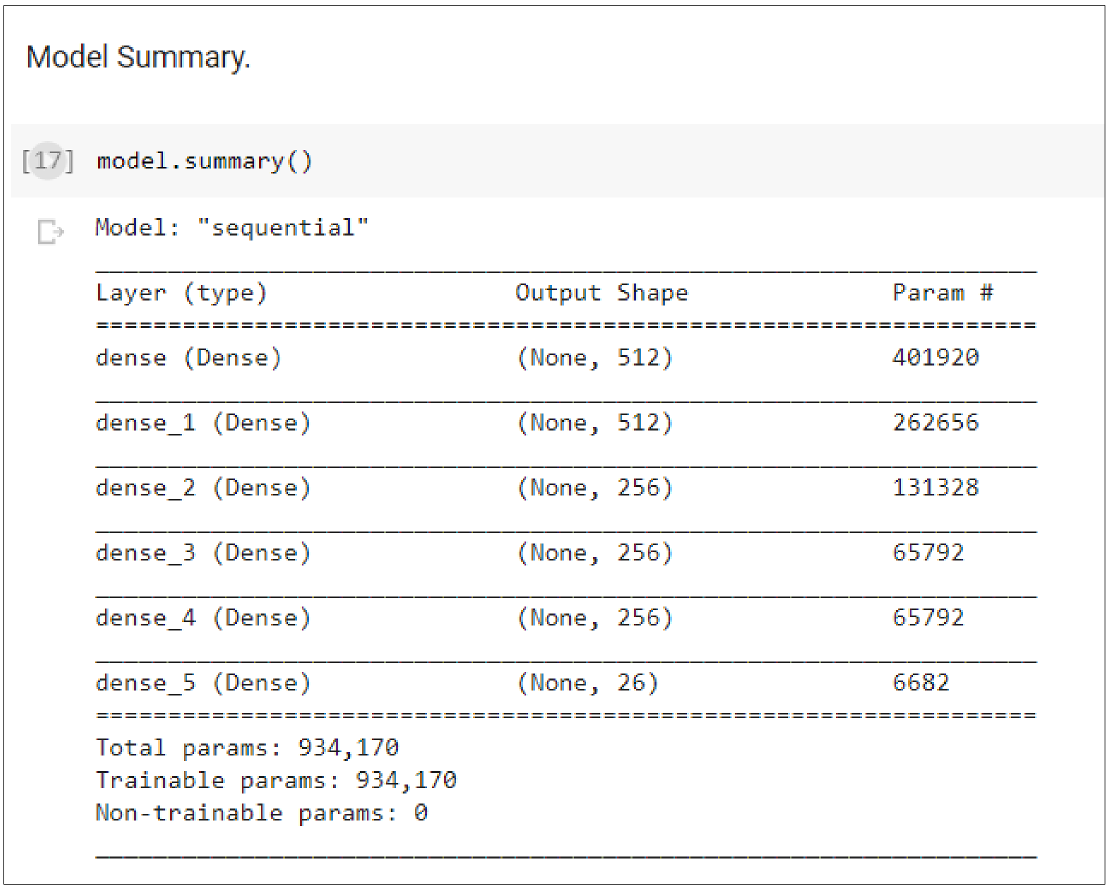






```
[16] from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense

     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 Model Summary:

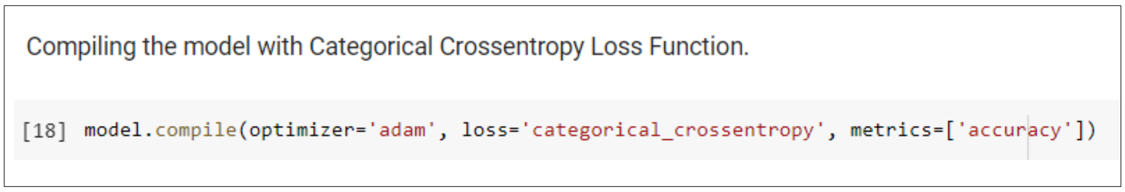


Model Summary.

```
[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
_________________________________________________________________
```

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



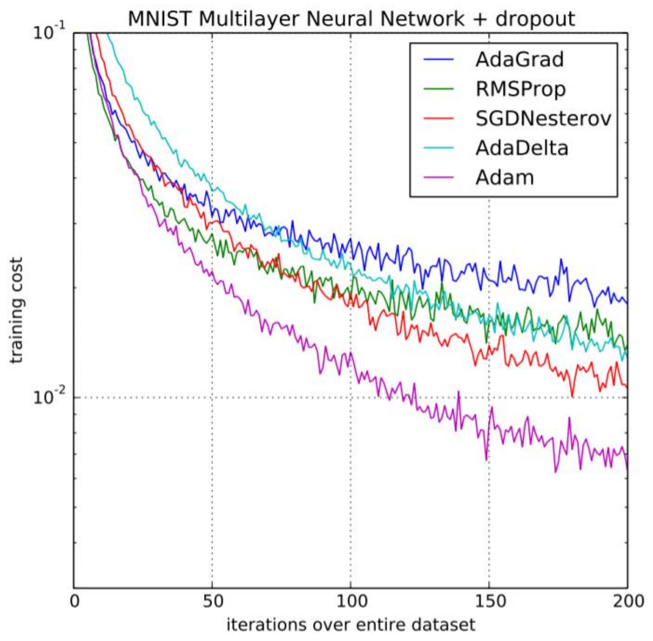
Compiling the model with Categorical Crossentropy Loss Function.

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

Adam Optimizer

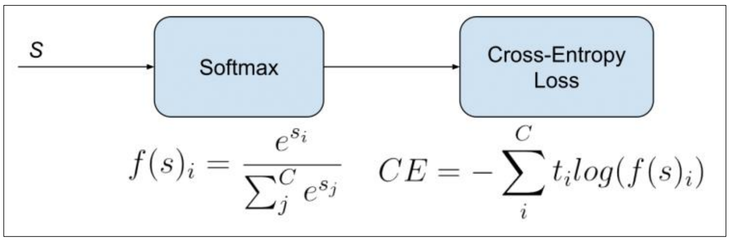
Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problem involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the "gradient descent with momentum" algorithm and the "RMSP" algorithm. Adam Optimizer inherits the strengths of the two methods and builds upon them to give a more optimized gradient descent.

The plot is shown below clearly depicts how Adam Optimizer outperforms the rest of the optimizer by a considerable margin in terms of training cost (low) and performance (high).



Categorical Crossentropy

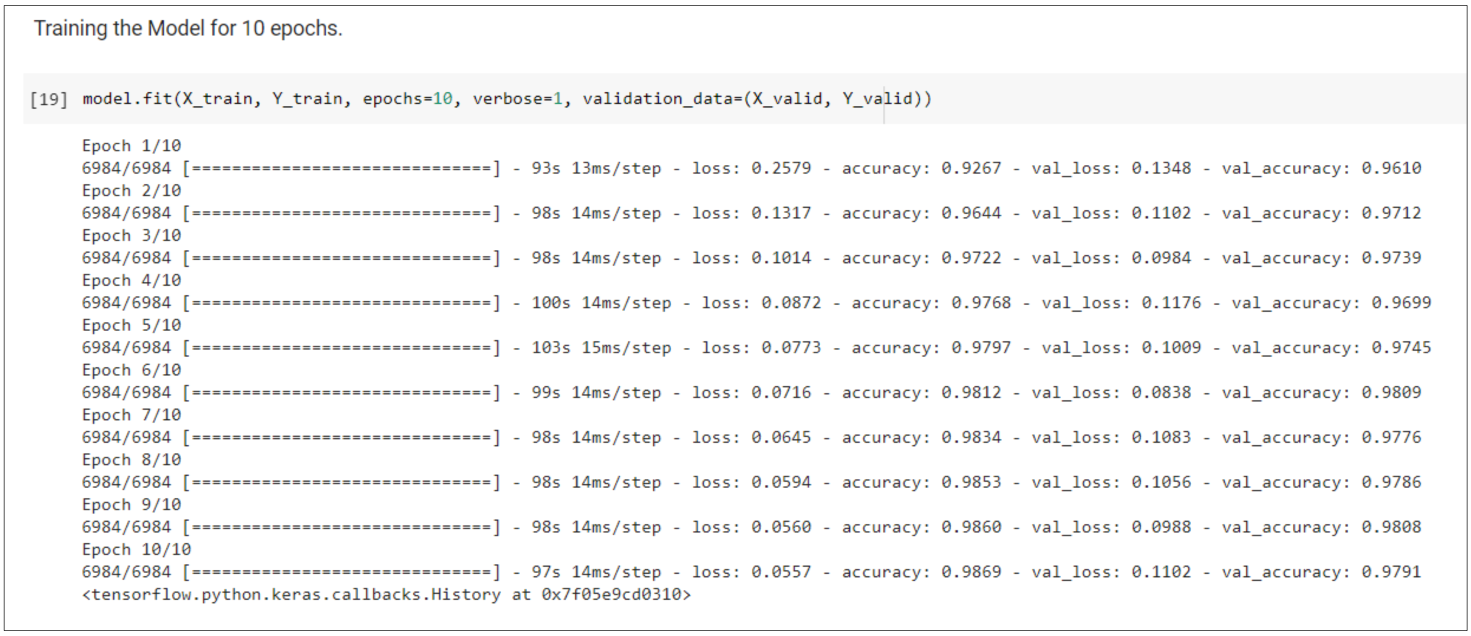
Also called Softmax Loss. It is a Softmax activation plus a Cross-Entropy loss. It is used for multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one.



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

After training the MLP model for 10 epochs, following are the metrics obtained:

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



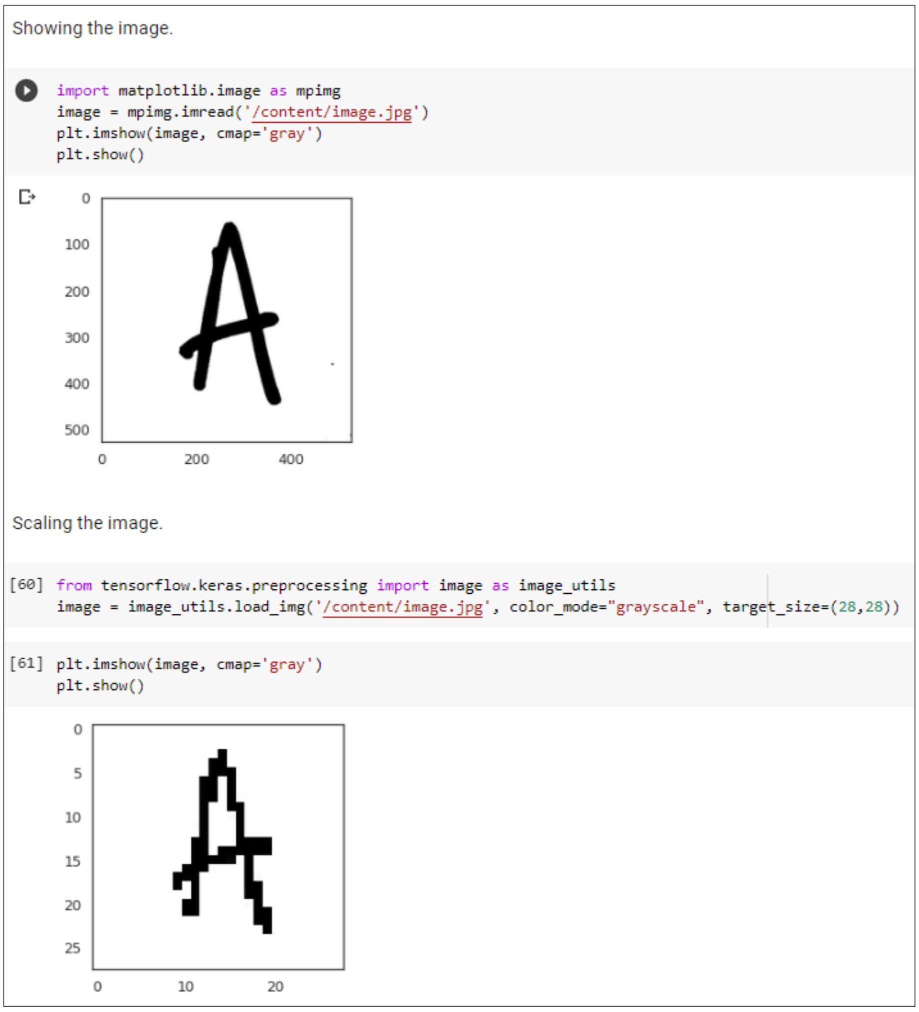
Training the Model for 10 epochs.

```
[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>
```

Predicting an image with the Model

Predicting the following image of the letter “A” (taken with my phone camera) using the model:

The images in the 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.



Showing the image.

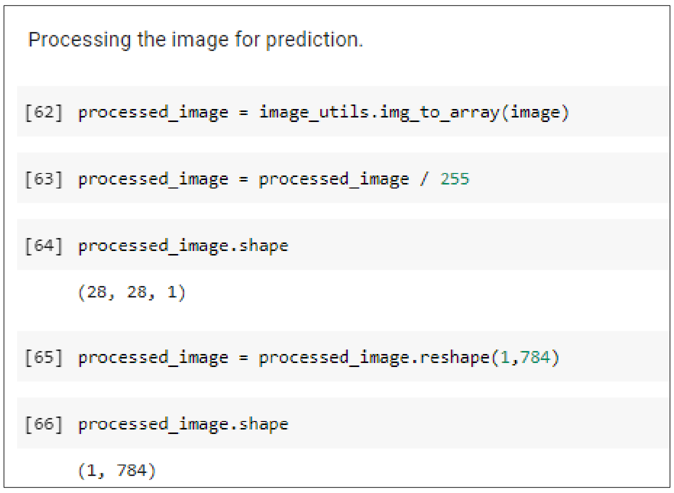
```
import matplotlib.image as mpimg
image = mpimg.imread('/content/image.jpg')
plt.imshow(image, cmap='gray')
plt.show()
```

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()
```

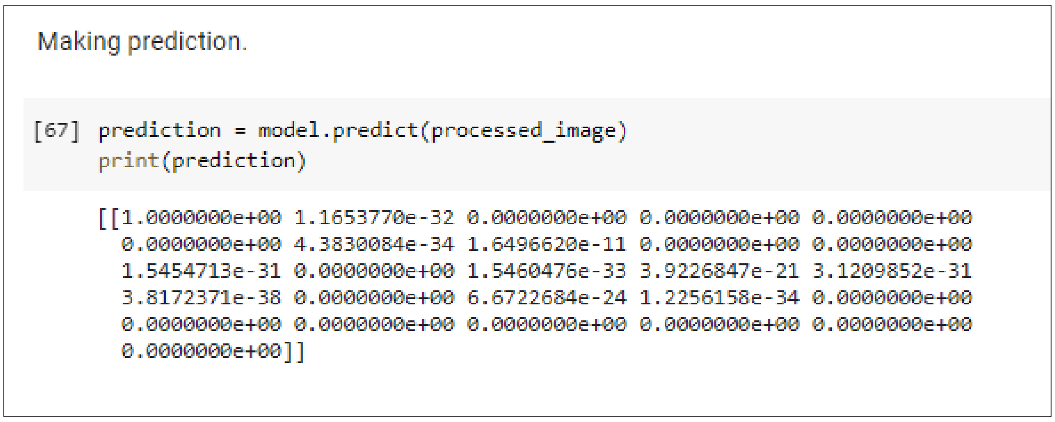
The image is converted into a more rudimentary format using the Keras utility called "image_to_array", after which it is reshaped to the correct shape.



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.

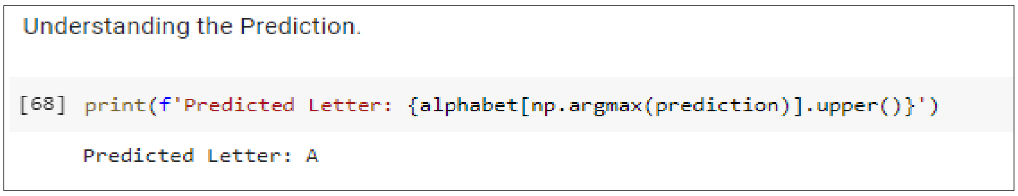


Making prediction.

```
[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 an array whose length is 26. 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" to determine the correct alphabet, where:

alphabet = "abcdefghijklmnopqrstuvwxyz"



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

A Handwritten Character Recognition (Text Recognition) MLP model has been successfully developed 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

1. https://data-flair.training/blogs/handwritten-character-recognition-neural-network/
2. https://www.kaggle.com/nohrud/99-9-accuracy-on-alphabet-recognition-by-eda
3. https://gombru.github.io/2018/05/23/cross_entropy_loss/
4. https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy
5. https://www.geeksforgeeks.org/intuition-of-adam-optimizer/
6. https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
7. https://en.wikipedia.org/wiki/Softmax_function
8. https://en.wikipedia.org/wiki/One-hot
9. https://en.wikipedia.org/wiki/Multilayer_perceptron
10. https://en.wikipedia.org/wiki/Handwriting_recognition
11. https://en.wikipedia.org/wiki/Optical_character_recognition