Deep Learning Mini-Project Report

on

Handwritten Character Recognition

By

Aditya Khandelwal(1032170752)
Aniket Kumar(1032171203)
Adesh Pawar(1032170104)
Angel Negi(1032170667)

Guided By

Prof. Anita Gunjal

Introduction

The handwritten character recognition is the ability of computers to recognize human handwritten characters/digits. It is a hard task for the machine because handwritten characters/digits are not perfect and can be made with many different flavors. The handwritten character recognition is the solution to this problem which uses the image of a character/digit and recognizes the same present in the image.

Dataset Used

1. MNIST:

    a. The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset.

    b. Size of Dataset : 122.2 MB

    c. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9.

    d. The set of images in the MNIST database was created in 1998 as a combination of two of NIST's databases

2. EMNIST:

   a. The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset . Further information on the dataset contents and conversion process can be found in the paper available at https://arxiv.org/abs/1702.05373v1

   b. There are six different splits provided in this dataset. A short summary of the dataset is provided below:

      i. EMNIST ByClass: 814,255 characters. 62 unbalanced classes.

      ii. EMNIST ByMerge: 814,255 characters. 47 unbalanced classes.

      iii. EMNIST Balanced: 131,600 characters. 47 balanced classes.

      iv. EMNIST Letters: 145,600 characters. 26 balanced classes.

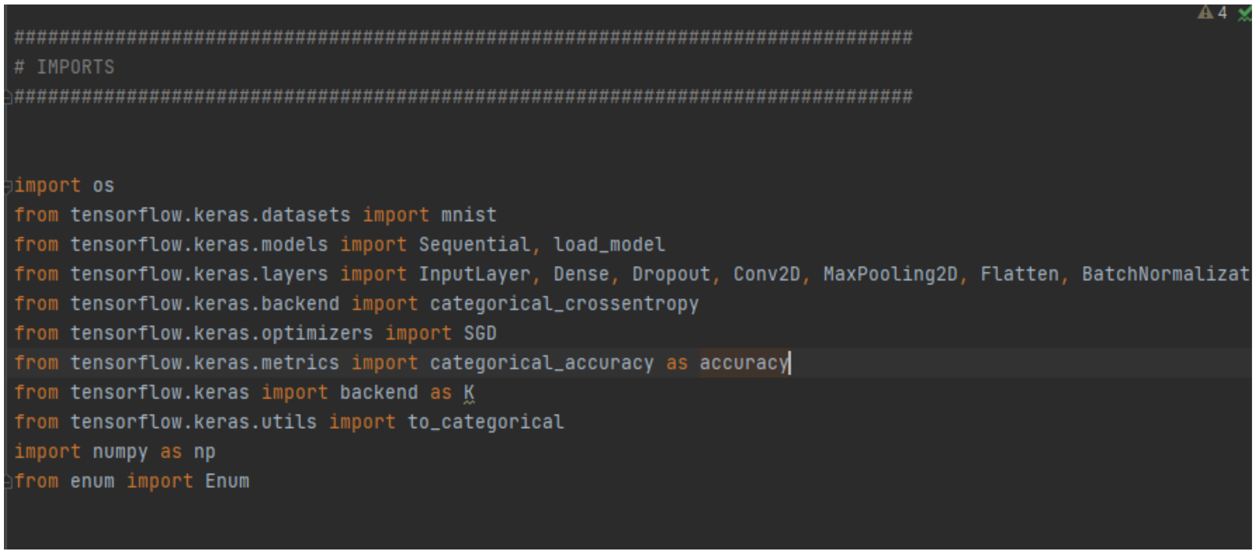
      v. EMNIST Digits: 280,000 characters. 10 balanced classes.

      vi. EMNIST MNIST: 70,000 characters. 10 balanced classes.

Methodology

1) Import the libraries and load the dataset

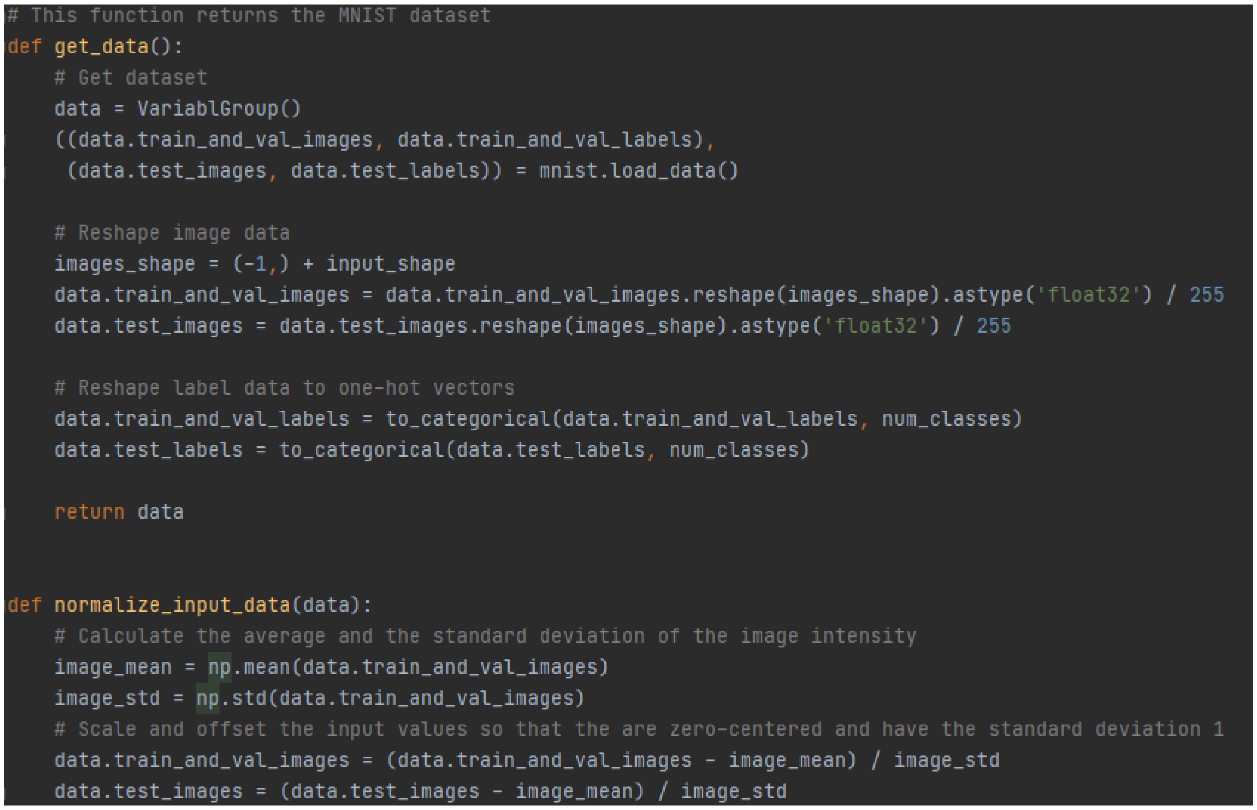
First, we are going to import all the modules that we are going to need for training our model. The Keras library already contains some datasets and MNIST is one of them. So we can easily import the dataset and start working with it. The mnist.load_data() method returns us the training data, its labels and also the testing data and its labels.



```
################################################################################
# IMPORTS
################################################################################


import os
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import InputLayer, Dense, Dropout, Conv2D, MaxPooling2D, Flatten, BatchNormalizat
from tensorflow.keras.backend import categorical_crossentropy
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.metrics import categorical_accuracy as accuracy
from tensorflow.keras import backend as K
from tensorflow.keras.utils import to_categorical
import numpy as np
from enum import Enum
```

2) Preprocess the data

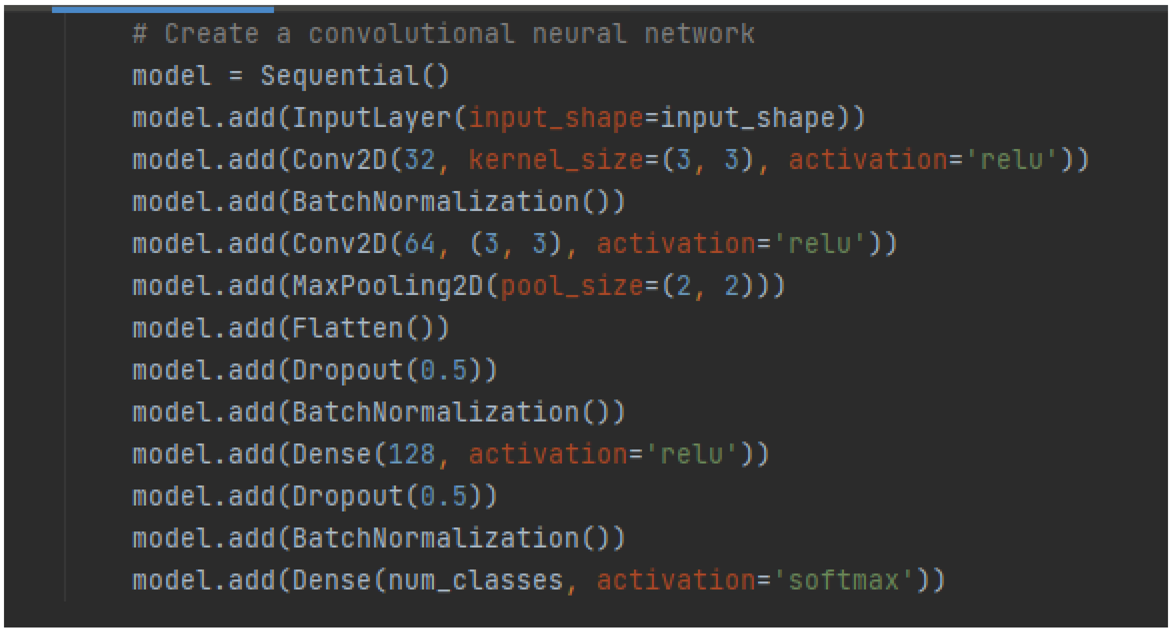
The image data cannot be fed directly into the model so we need to perform some operations and process the data to make it ready for our neural network. The dimension of the training data is (60000,28,28). The CNN model will require one more dimension so we reshape the matrix to shape (60000,28,28,1).



```
# This function returns the MNIST dataset
def get_data():
    # Get dataset
    data = VariablGroup()
    ((data.train_and_val_images, data.train_and_val_labels),
     (data.test_images, data.test_labels)) = mnist.load_data()

    # Reshape image data
    images_shape = (-1,) + input_shape
    data.train_and_val_images = data.train_and_val_images.reshape(images_shape).astype('float32') / 255
    data.test_images = data.test_images.reshape(images_shape).astype('float32') / 255

    # Reshape label data to one-hot vectors
    data.train_and_val_labels = to_categorical(data.train_and_val_labels, num_classes)
    data.test_labels = to_categorical(data.test_labels, num_classes)

    return data


def normalize_input_data(data):
    # Calculate the average and the standard deviation of the image intensity
    image_mean = np.mean(data.train_and_val_images)
    image_std = np.std(data.train_and_val_images)
    # Scale and offset the input values so that the are zero-centered and have the standard deviation 1
    data.train_and_val_images = (data.train_and_val_images - image_mean) / image_std
    data.test_images = (data.test_images - image_mean) / image_std
```

3) Create the Model

Now we will create our CNN model in Python. A CNN model generally consists of convolutional and pooling layers. It works better for data that are represented as grid structures, this is the reason why CNN works well for image classification problems. The dropout layer is used to deactivate some of the neurons and while training, it reduces offer fitting of the model.

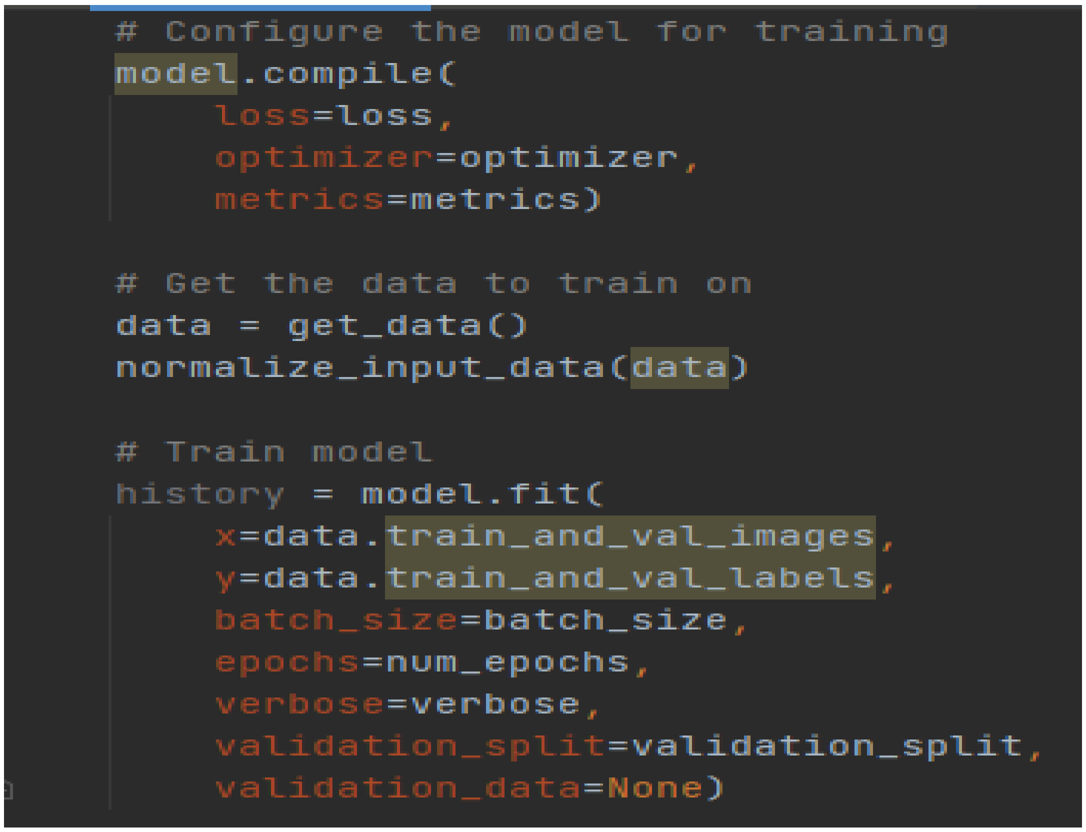


```python
# Create a convolutional neural network
model = Sequential()
model.add(InputLayer(input_shape=input_shape))
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(num_classes, activation='softmax'))
```

4) Training of Model

The model.fit() function of Keras will start the training of the model. It takes the training data, validation data, epochs, and batch size.

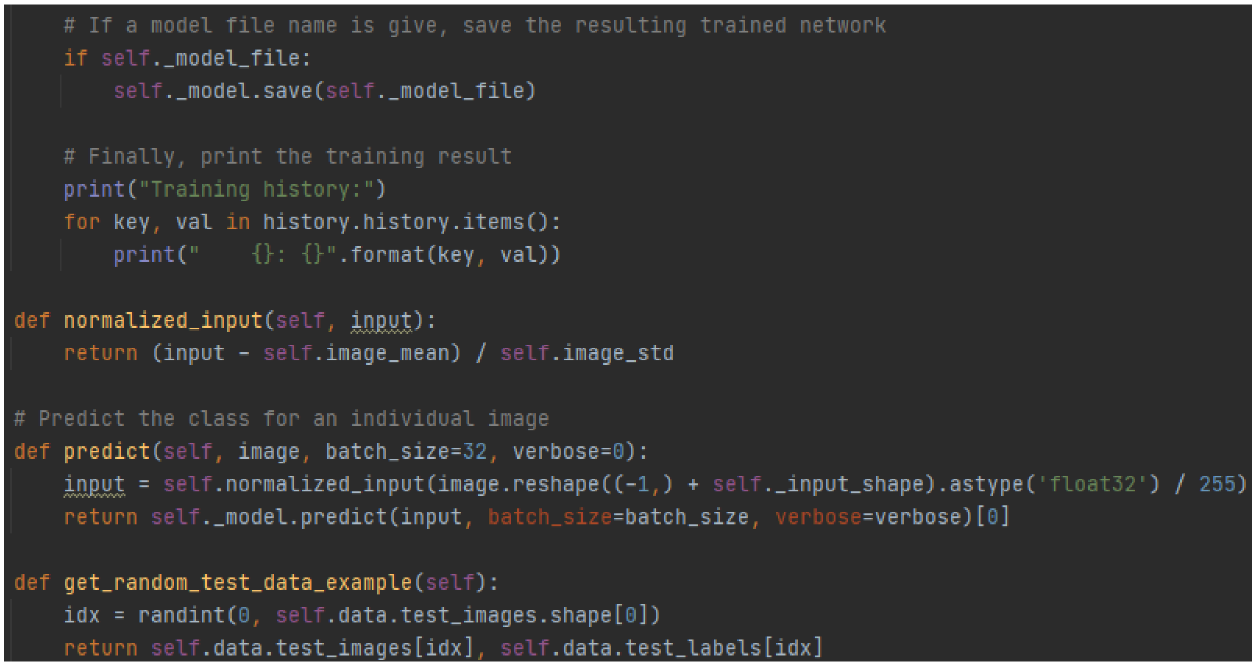
It takes some time to train the model. After training, we save the weights and model definition in the 'mnist.h5' file.



```
# Configure the model for training
model.compile(
    loss=loss,
    optimizer=optimizer,
    metrics=metrics)

# Get the data to train on
data = get_data()
normalize_input_data(data)

# Train model
history = model.fit(
    x=data.train_and_val_images,
    y=data.train_and_val_labels,
    batch_size=batch_size,
    epochs=num_epochs,
    verbose=verbose,
    validation_split=validation_split,
    validation_data=None)
```

5) Evaluate the Model

We have 10,000 images in our dataset which will be used to evaluate how good our model works. The testing data was not involved in the training of the data therefore, it is new data for our model. The MNIST dataset is well balanced so we can get around 99% accuracy.

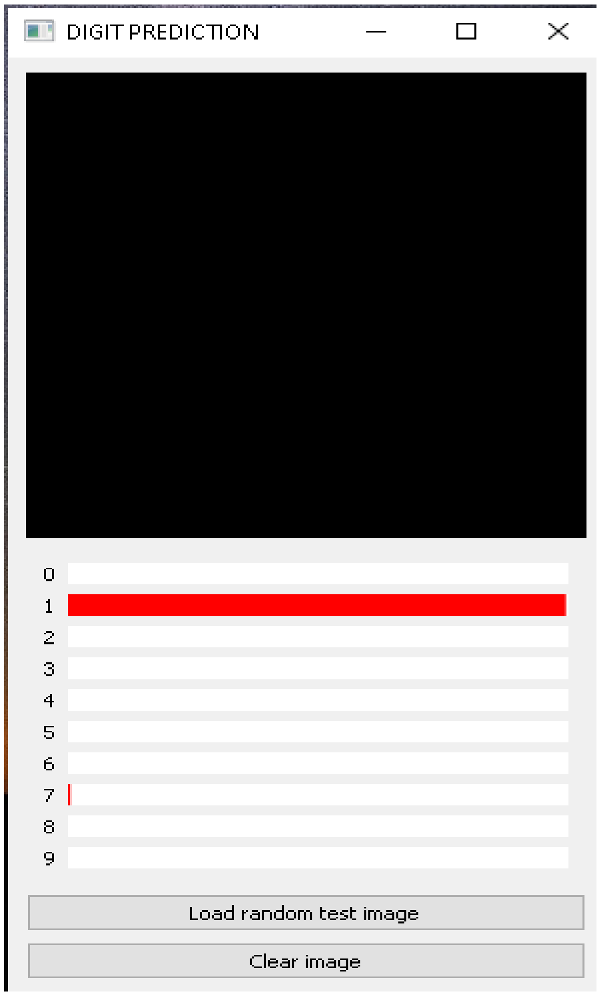


```
    # If a model file name is give, save the resulting trained network
    if self._model_file:
        self._model.save(self._model_file)

    # Finally, print the training result
    print("Training history:")
    for key, val in history.history.items():
        print("    {}: {}".format(key, val))

def normalized_input(self, input):
    return (input - self.image_mean) / self.image_std

# Predict the class for an individual image
def predict(self, image, batch_size=32, verbose=0):
    input = self.normalized_input(image.reshape((-1,) + self._input_shape).astype('float32') / 255)
    return self._model.predict(input, batch_size=batch_size, verbose=verbose)[0]

def get_random_test_data_example(self):
    idx = randint(0, self.data.test_images.shape[0])
    return self.data.test_images[idx], self.data.test_labels[idx]
```

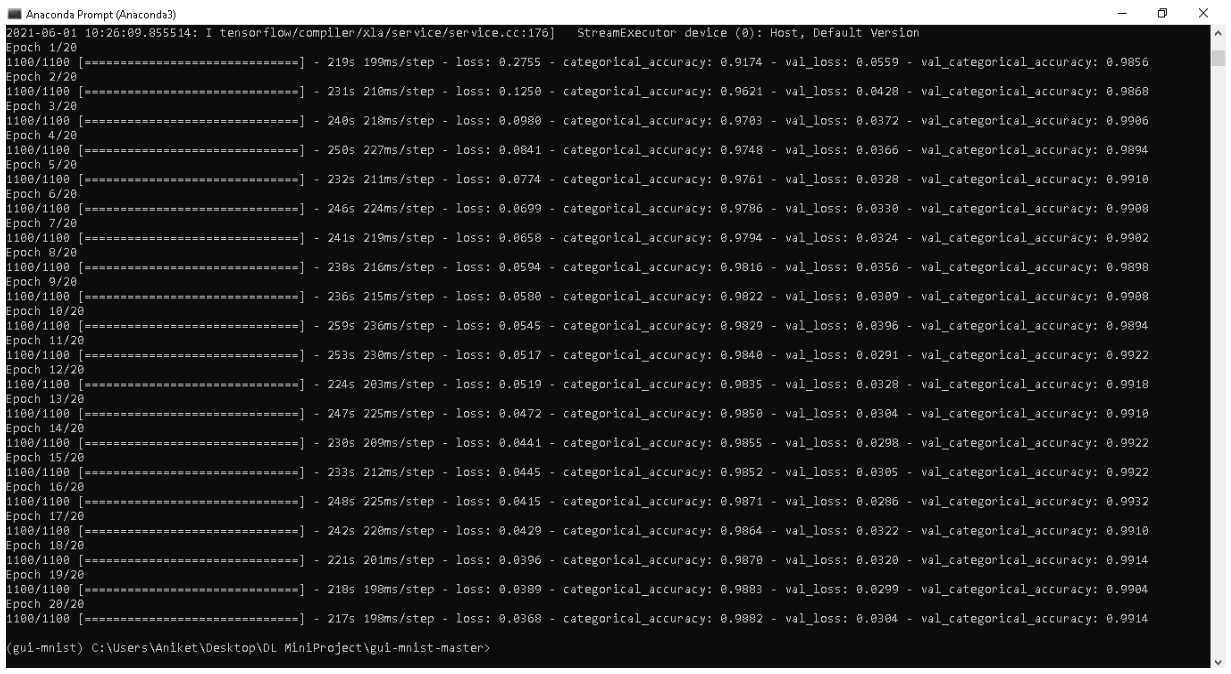
6) Create GUI to predict Characters/Digits

Now for the GUI, we have created a new file in which we build an interactive window to draw digits on canvas and with a button, we can recognize the digit. The PySide2 library comes in the Python standard library. We have created a function predict_digit() that takes the image as input and then uses the trained model to predict the digit.

Then we create the App class which is responsible for building the GUI for our app. We create a canvas where we can draw by capturing the mouse event and with a button, we trigger the predict_digit() function and display the results.

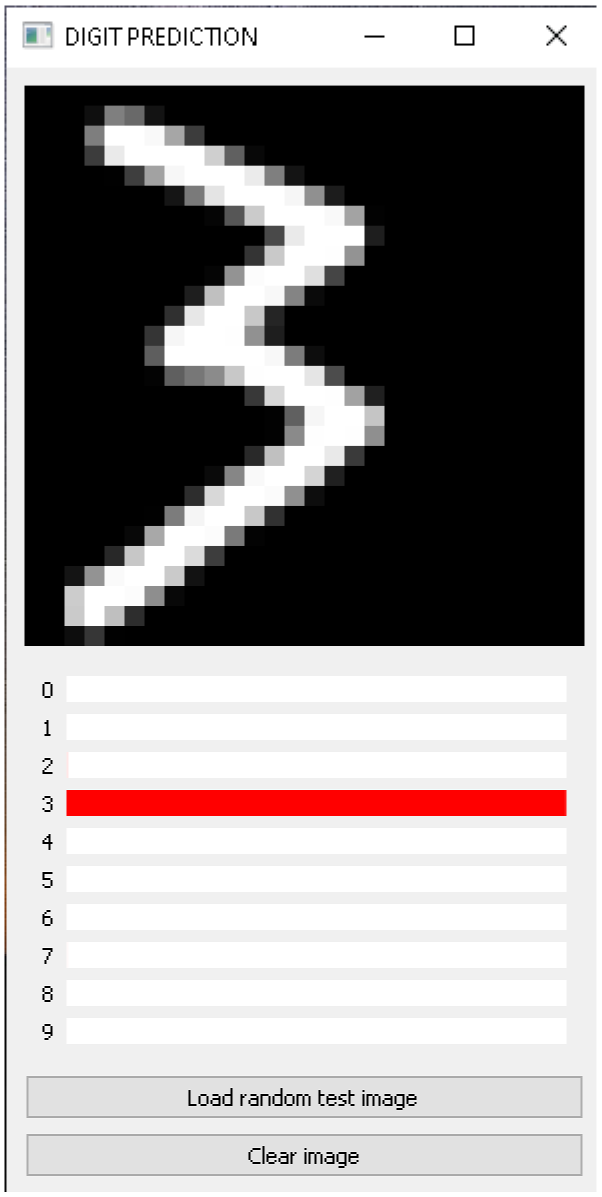


Output



Anaconda Prompt (Anaconda3)

```
2021-06-01 10:26:09.855514: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
Epoch 1/20
1100/1100 [==============================] - 219s 199ms/step - loss: 0.2755 - categorical_accuracy: 0.9174 - val_loss: 0.0559 - val_categorical_accuracy: 0.9856
Epoch 2/20
1100/1100 [==============================] - 231s 210ms/step - loss: 0.1250 - categorical_accuracy: 0.9621 - val_loss: 0.0428 - val_categorical_accuracy: 0.9868
Epoch 3/20
1100/1100 [==============================] - 240s 218ms/step - loss: 0.0980 - categorical_accuracy: 0.9703 - val_loss: 0.0372 - val_categorical_accuracy: 0.9906
Epoch 4/20
1100/1100 [==============================] - 250s 227ms/step - loss: 0.0841 - categorical_accuracy: 0.9748 - val_loss: 0.0366 - val_categorical_accuracy: 0.9894
Epoch 5/20
1100/1100 [==============================] - 232s 211ms/step - loss: 0.0774 - categorical_accuracy: 0.9761 - val_loss: 0.0328 - val_categorical_accuracy: 0.9910
Epoch 6/20
1100/1100 [==============================] - 246s 224ms/step - loss: 0.0699 - categorical_accuracy: 0.9786 - val_loss: 0.0330 - val_categorical_accuracy: 0.9908
Epoch 7/20
1100/1100 [==============================] - 241s 219ms/step - loss: 0.0658 - categorical_accuracy: 0.9794 - val_loss: 0.0324 - val_categorical_accuracy: 0.9902
Epoch 8/20
1100/1100 [==============================] - 238s 216ms/step - loss: 0.0594 - categorical_accuracy: 0.9816 - val_loss: 0.0356 - val_categorical_accuracy: 0.9898
Epoch 9/20
1100/1100 [==============================] - 236s 215ms/step - loss: 0.0580 - categorical_accuracy: 0.9822 - val_loss: 0.0309 - val_categorical_accuracy: 0.9908
Epoch 10/20
1100/1100 [==============================] - 259s 236ms/step - loss: 0.0545 - categorical_accuracy: 0.9829 - val_loss: 0.0396 - val_categorical_accuracy: 0.9894
Epoch 11/20
1100/1100 [==============================] - 253s 230ms/step - loss: 0.0517 - categorical_accuracy: 0.9840 - val_loss: 0.0291 - val_categorical_accuracy: 0.9922
Epoch 12/20
1100/1100 [==============================] - 224s 203ms/step - loss: 0.0519 - categorical_accuracy: 0.9835 - val_loss: 0.0328 - val_categorical_accuracy: 0.9918
Epoch 13/20
1100/1100 [==============================] - 247s 225ms/step - loss: 0.0472 - categorical_accuracy: 0.9850 - val_loss: 0.0304 - val_categorical_accuracy: 0.9910
Epoch 14/20
1100/1100 [==============================] - 230s 209ms/step - loss: 0.0441 - categorical_accuracy: 0.9855 - val_loss: 0.0298 - val_categorical_accuracy: 0.9922
Epoch 15/20
1100/1100 [==============================] - 233s 212ms/step - loss: 0.0445 - categorical_accuracy: 0.9852 - val_loss: 0.0305 - val_categorical_accuracy: 0.9922
Epoch 16/20
1100/1100 [==============================] - 248s 225ms/step - loss: 0.0415 - categorical_accuracy: 0.9871 - val_loss: 0.0286 - val_categorical_accuracy: 0.9932
Epoch 17/20
1100/1100 [==============================] - 242s 220ms/step - loss: 0.0429 - categorical_accuracy: 0.9864 - val_loss: 0.0322 - val_categorical_accuracy: 0.9910
Epoch 18/20
1100/1100 [==============================] - 221s 201ms/step - loss: 0.0396 - categorical_accuracy: 0.9870 - val_loss: 0.0320 - val_categorical_accuracy: 0.9914
Epoch 19/20
1100/1100 [==============================] - 218s 198ms/step - loss: 0.0389 - categorical_accuracy: 0.9883 - val_loss: 0.0299 - val_categorical_accuracy: 0.9904
Epoch 20/20
1100/1100 [==============================] - 217s 198ms/step - loss: 0.0368 - categorical_accuracy: 0.9882 - val_loss: 0.0304 - val_categorical_accuracy: 0.9914

(gui-mnist) C:\Users\Aniket\Desktop\DL MiniProject\gui-mnist-master>
```


DIGIT PREDICTION
0
1
2
3
4
5
6
7
8
9
Load random test image
Clear image

Conclusion

The main aim of this project is to find a representation of handwritten digits that allow their effective recognition. In this project, we used different machine learning algorithm for recognition of handwritten numerals. In any recognition process, the important problem is to address the feature extraction and correct classification approaches. The proposed algorithm tries to address both the factors and well in terms of accuracy and time complexity. The overall highest accuracy 98.82% is achieved in the recognition process by Multilayer Perceptron.