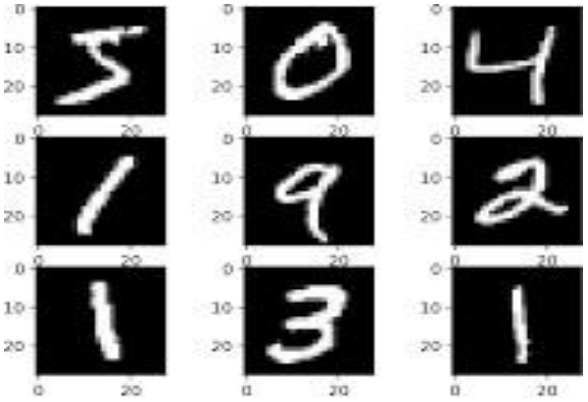
COMPUTER VISION CEP REPORT

"HANDWRITTEN DIGITS RECOGNITION USING CNN"



GROUP MEMBERS	*ROLL NO*
LABIQA SIDDIQUI	*CS-19066*
JAVERIA ADIL	*CS-19301*
MUSFIRAH FAYYAZ	*CS-19303*

SUBMITTED TO MISS FAUZIA YASIR

SUBMISSION DATE:14-06-2023

INTRODUCTION

MNIST dataset has been used in this project for the implementation of a handwritten digit recognition with 10-digit classes from (0-9). To implement this, a special type of deep neural network called Convolutional Neural Networks (CNN) has been used. In the end, we have also built a Graphical user interface (GUI) where we can directly test it by drawing the digit and recognizing it straight away.

What is Handwritten Digit Recognition?

Handwritten digit recognition is the process to provide the ability to machines to recognize human handwritten digits. It is not an easy task for the machine because handwritten digits are not perfect, vary from person-to-person, and can be made with many different flavors.

Convolutional Neural Networks (CNN)

CNN is a deep learning technique to classify the input automatically. It is a reliable deep learning algorithm for an automated end-to-end prediction. CNN essentially extracts 'useful' features from the given input automatically making it super easy for us!

A CNN model consists of three primary layers: Convolutional Layer, Pooling layer(s), and fully connected layer.

(1) Convolutional Layer: This layer **extracts high-level input features** from input data and passes those features to the next layer in the form of feature maps.

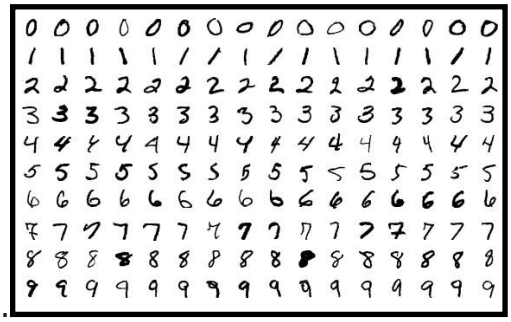
(2) Pooling Layer: It is used **to reduce the dimensions of data** by applying pooling on the feature map to generate new feature maps with reduced dimensions.

(3) Fully-Connected Layer: Finally, the classification is done by this layer. Probability scores are calculated for each class label by a popular activation function called the **softmax function**.

DATASET

THE MNIST CLASSIFICATION DATASET of handwritten digits is the most popular dataset. This dataset consists of 28x28 grayscale 70,000 images of handwritten 10 digits from 0-9. Above 60,000 plus training images of handwritten digits from 0 to 9 and more than 10,000 images for testing are present in the MNIST dataset. So, 10 different classes are in the MNIST dataset. The images of handwritten digits are shown as a matrix of 28×28 where every cell consists of a grayscale pixel value.

Keras is a deep learning API written in Python and MNIST is a dataset provided by this API. When the Keras API is called, there are four values returned namely



CODE EXPLANATION

IMPORTING LIBRARIES & DATASET

At the project beginning, we import all the needed modules for training our model.

IMPORTING THE LIBRARIES

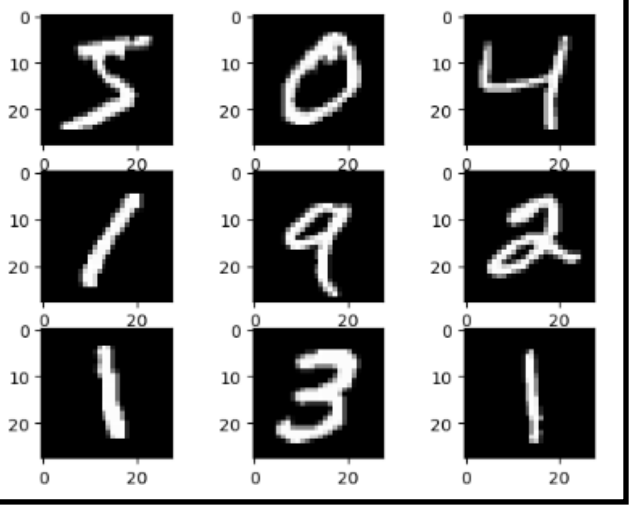
```
import tensorflow as tf #TF is a free and open-sou
import keras #TF already contain data which can be
import cv2 #CV to load & process images
import numpy as np #python library important for w
import matplotlib.pyplot as plt #plotting library
from keras.datasets import mnist # importing datas
from keras.models import Sequential #importing seq
from keras.layers import Dense, Dropout, Flatten #
from keras.layers import Conv2D, MaxPooling2D #imp
from keras import backend as K #importing backend
```

LOADING THE DATASET

We loaded the MNIST dataset by calling the **`mnist.load_data()`** function to get training & testing data with their labels. This dataset returns four values i-e x_train, y_train, x_test, and y_test. Where x_train and x_test contain the images, and y_train and y_test contain the digits that those images represent.

SPLITTING THE MNIST DATA IN TO TRAIN AND TEST DATASETS

```
In [3]: (x_train, y_train), (x_test, y_test) = mnist.load_data()  #unpacking

In [4]: #plot first few images in the dataset is also created showing the nu
        for i in range(9):
            plt.subplot(330 + 1 + i)
            plt.imshow(x_train[i], cmap=plt.get_cmap('gray'))
        # show the figure
        plt.show()
```



```
In [5]: print(x_train.shape, y_train.shape) #T
        (60000, 28, 28) (60000,)

In [6]: x_train.shape #finding train data
Out[6]: (60000, 28, 28)

In [7]: x_test.shape #finding test data
Out[7]: (10000, 28, 28)
```

x_train: uint8 NumPy array of grayscale image data with shapes (60000, 28, 28), containing the training data. Pixel values range from 0 to 255.

y_train: uint8 NumPy array of digit labels (integers in range 0-9) with shape (60000,) for the training data.

x_test: uint8 NumPy array of grayscale image data with shapes (10000, 28, 28), containing the test data. Pixel values range from 0 to 255.

y_test: uint8 NumPy array of digit labels (integers in range 0-9) with shape (10000,) for the test data.

DATA PRE-PROCESSING

Data has to be processed, cleaned, rectified in order to improve its quality. CNN will learn best from a dataset that does not contain any null values, has all numeric data, and is scaled. So, here we have performed some steps to ensure that our dataset is perfectly suitable for a CNN model.

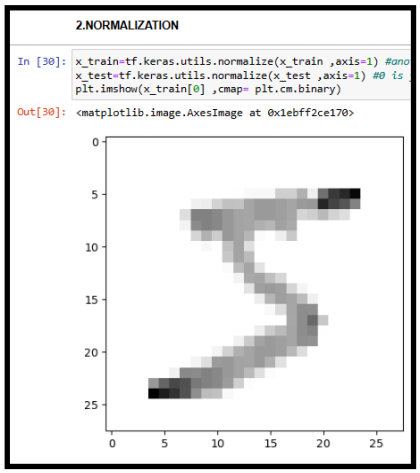
For Example

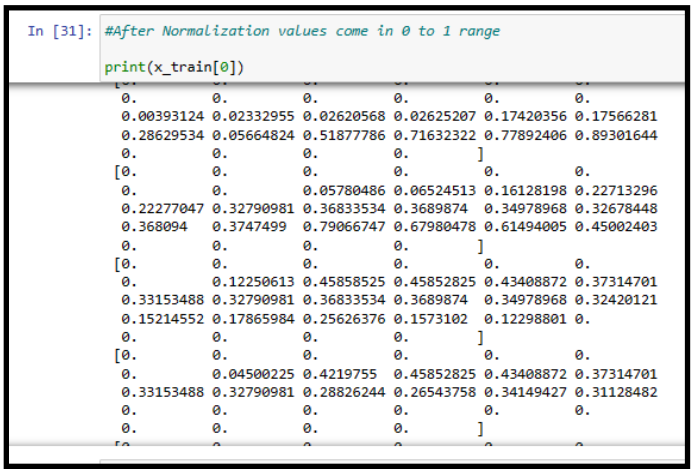
If you write X_train[0] then you get the 0th image with values between 0-255 (0 means black and 255 means white). The output is a 2-dimensional matrix (Of course, we will not know what handwritten digit X_train[0] represents. To know this write y_train[0] and you will get 5 as output. This means that the 0th image of this training dataset represents the number 5.

1. NORMALIZATION

Data normalization has been performed on the training and testing images using **`normalize()`** function. This function is used to scale the pixel values of the images between 0 and 1. Dividing each pixel value by the maximum pixel value, to ensure that the pixel values are in a consistent and standardized range. By normalizing the data, the neural network can learn more effectively, as it can prevent issues of different scales or ranges of the input features.

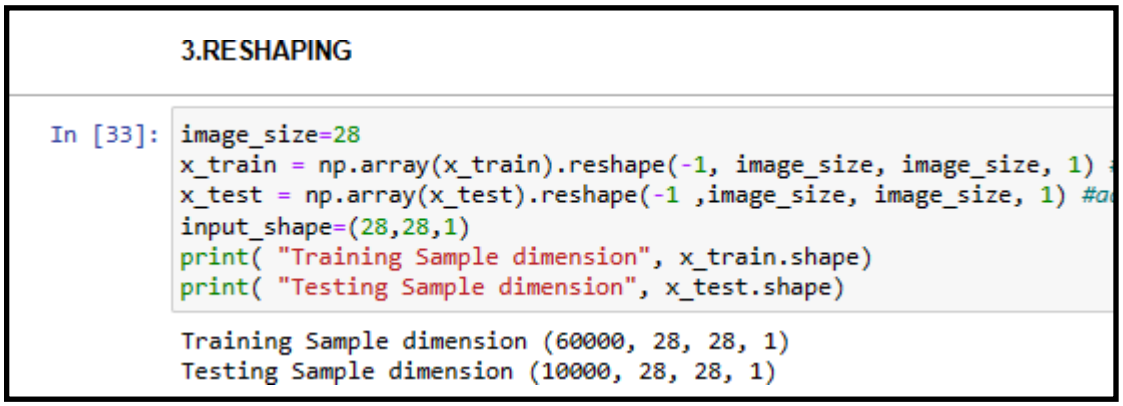
Setting **`axis=1`** means that the normalization is performed along the rows of the image array. After normalizing the data, we displayed the first image from the normalized training set with a binary colormap (**`cmap=plt.cm.binary`**). This allows visualizing the normalized image in black and white. Then we print the normalized pixel values of the first image in the training set which displays the array of pixel values after normalization.





2. RESHAPING

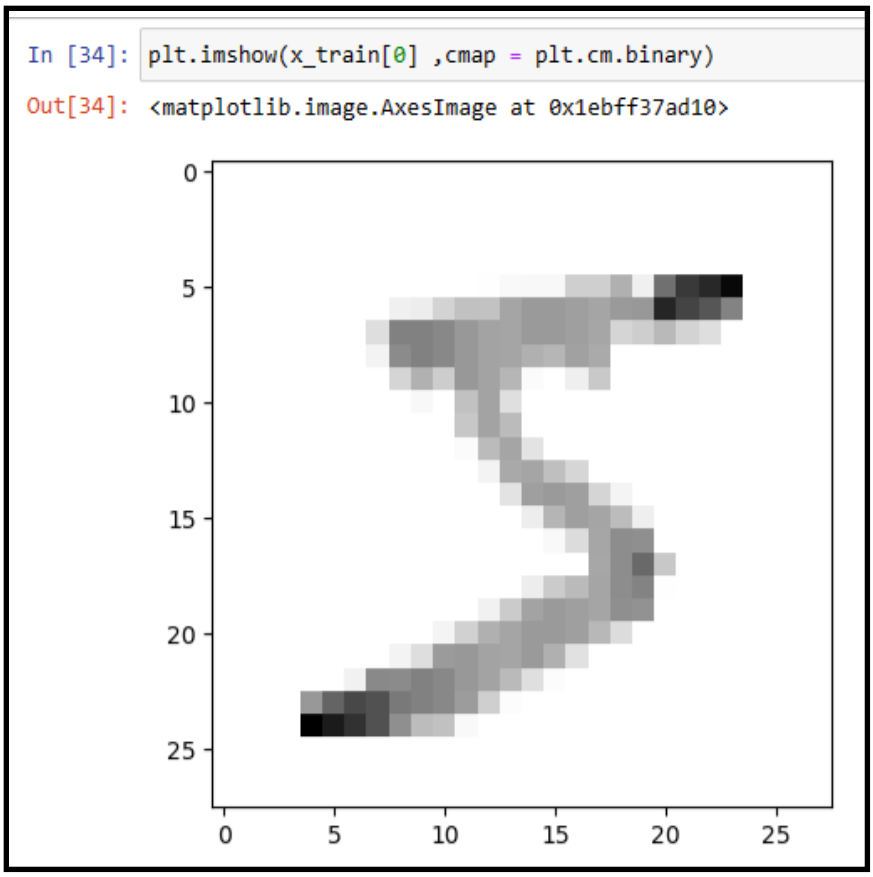
Reshaping has been performed on training and testing images to ensure that they have the correct dimensions and are compatible with the CNN model. In the MNIST dataset, each image is originally a 2-dimensional array of size 28x28, representing a grayscale image. However, for CNN models, the input is typically expected to have an additional dimension for the channel.



where `np.array(x_train)` converts the x_train list into a NumPy array. `. reshape(-1, image_size, image_size, 1)` reshapes the array to the desired shape. When using the -1 argument in the reshape function, it means that the size of that particular dimension is automatically determined by the reshape operation itself. The `image_size` arguments define the height and width dimensions of the images, and `1` represents the grayscale channel.

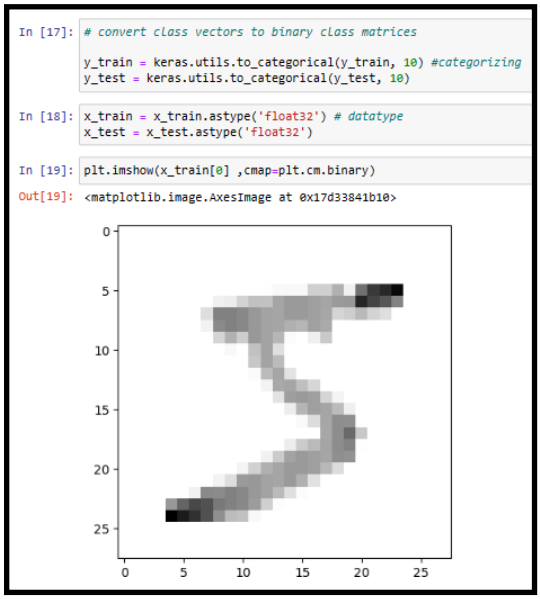
Similarly, the same reshaping operation is performed on x_test to ensure consistency in dimensions. Then resulting shapes of the reshaped arrays are printed. Finally, the code displays the first image from the reshaped training set with the binary colormap **`cmap=plt.cm.binary`**.

This reshaping is necessary to match the expected input shape of the CNN model.



3. LABEL ENCODING

Then we performed additional preprocessing steps on the target labels (y_train and y_test) and the input images (x_train and x_test) before training the model to ensure compatibility with the model architecture and to improve the convergence and performance of the model.



`keras.utils.to_categorical(y_train, 10)` **&** **`keras.utils.to_categorical(y_test, 10)`** convert the target labels (y_train and y_test) into one-hot encoded vectors. One-hot encoding is a process of representing categorical data in a binary format, where each category is represented by a binary vector with a single 1 and the rest 0s. The `10` as the second argument specifies the number of classes in the dataset. Since it is handwritten digits recognition, there are 10 classes representing digits from 0 to 9.

`x_train=x_train.astype('float32')` and **`x_test=x_test.astype('float32')`** convert the input images (x_train and x_test) to the data type float32. It is a common practice to normalize the pixel values of the images by scaling them to the range [0, 1]. The `astype('float32')` converts the pixel values from integers to floating-point numbers, allowing the division by 255.0 in the normalization step.

MODEL CREATION

Next, we created the architecture of the CNN model using the Keras Sequential API and configured the model for training. A convolutional layer and pooling layers are the two wheels of a CNN model.

CREATING A DEEP NEURAL NETWORK

APPLYING MODEL

In [20]:
```
batch_size = 128
num_classes = 10
epochs = 10
model = Sequential()

model.add(Conv2D(25, kernel_size=(4, 4),activation='relu',input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
#Output
model.add(Dense(num_classes, activation='softmax'))

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

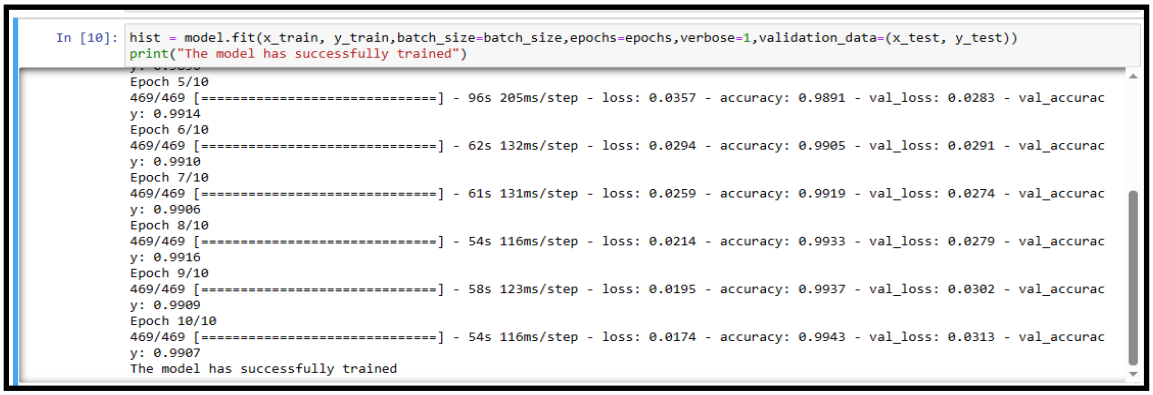
OPTIMIZATION

Previously we used Adadelta optimizer which degraded was degrading our performance then we try different optimizers to fix this issue. Finally, adam works the best among all.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
#model.compile(loss=keras.losses.categorical_crossentropy,optimizer=keras.optimizers.Adadelta(),metrics=['accuracy'])
```

TRAINING

Then our code trains the defined CNN model using the training data (x_train and y_train) and evaluates its performance on the validation data (x_test and y_test).



In the CNN model, there is an input layer followed by two hidden layers and finally an output layer. In the most simplest terms, activation functions are responsible for making decisions of whether or not to move forward.

Once the model has been created, it is time to compile it and fit the model. During the process of fitting, the model will go through the dataset and understand the relations. It will learn throughout the process as many times as has been defined. In our example, we have defined 10 epochs.

During the process, the CNN model will learn and also make mistakes. For every mistake (i.e., wrong predictions), there is a penalty and that is represented in the loss value for each epoch. In short, the model should generate as little loss and as high accuracy as possible at the end of the last epoch.

EVALUATION

Then we evaluated the trained model's performance on the test data (x_test and y_test) and prints the test loss and test accuracy. By evaluating the model on the test data and printing the test loss and test accuracy, you can assess how well the trained model generalizes to new, unseen data and evaluate its overall performance.

```
In [13]: score = model.evaluate(x_test, y_test, verbose=0)
         print('Test loss:', score[0])
         print('Test accuracy:', score[1])

         Test loss: 0.03125268593430519
         Test accuracy: 0.9907000064849854
```

We have used our test dataset to see how well the CNN model will perform.

GUI CREATION FOR DIGIT PREDICTION

This code imports necessary libraries & modules for a specific task. These import statements make the necessary libraries and modules available for use in the code that follows, enabling functions related to loading pre-trained models, creating GUIs.

GUI INTERFACE FOR PREDICTING DIGIT

```
In [ ]: from keras.models import load_model
        from tkinter import *
        import tkinter as tk
        from PIL import ImageGrab, Image
        import numpy as np
```

`predict_digit()` function takes an image as input and returns the predicted digit label and the confidence score for that prediction.

```
In [ ]: def predict_digit(img):
            #resize image to 28x28 pixels
            img = img.resize((28,28))
            #convert rgb to grayscale
            img = img.convert('L')
            img = np.array(img)
            #reshaping to support our model
            img = img.reshape(1,28,28,1)
            img = img/255.0
            #predicting the class
            res = model.predict([img])[0]
            return np.argmax(res), max(res)
```

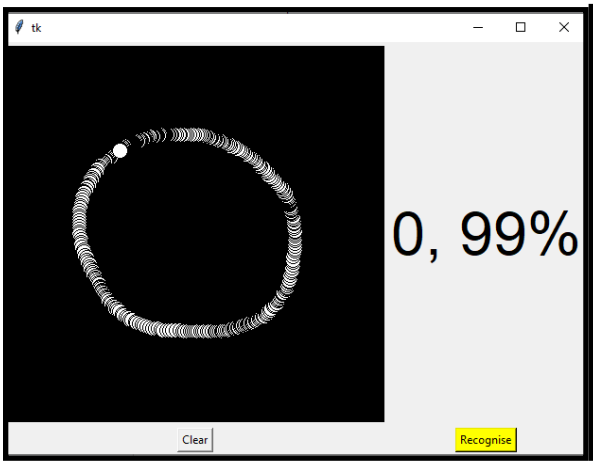
App class inherits from the tkinter.Tk class. This class represents a GUI application for handwriting recognition which sets up the GUI application with a canvas to draw on, buttons for recognition and clearing, and functionality to handle drawing and recognizing the handwritten digits.

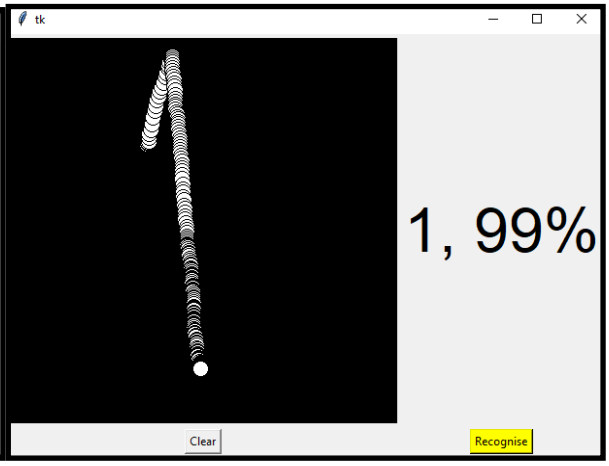
```
In [ ]: class App(tk.Tk):
            def __init__(self):
                tk.Tk.__init__(self)
                self.x = self.y = 0

                # Creating elements
                self.canvas = tk.Canvas(self, width=400, height=400, bg = "white", cursor="cross")
                self.label = tk.Label(self, text="Draw..", font=("Helvetica", 48))
                self.classify_btn = tk.Button(self, text = "Recognise", command = self.classify_handwriting, bg="yellow")
                self.button_clear = tk.Button(self, text = "Clear", command = self.clear_all)

                # Grid structure
                self.canvas.grid(row=0, column=0, pady=2, sticky=W, )
                self.label.grid(row=0, column=1,pady=2, padx=2)
                self.classify_btn.grid(row=1, column=1, pady=2, padx=2)
                self.button_clear.grid(row=1, column=0, pady=2)

                #self.canvas.bind("<Motion>", self.start_pos)
                self.canvas.bind("<B1-Motion>", self.draw_lines)

            def clear_all(self):
                self.canvas.delete("all")

            def classify_handwriting(self):
                HWND = self.canvas.winfo_id()  # get the handle of the canvas
                x, y = (self.canvas.winfo_rootx(), self.canvas.winfo_rooty())# get the coordinate of the canvas
                width, height = (self.canvas.winfo_width(), self.canvas.winfo_height())

                a, b, c, d = (x, y, x+width, y+height)
                rect=(a,b,c,d)
                im = ImageGrab.grab(rect)
                digit, acc = predict_digit(im)
                self.label.configure(text= str(digit)+', '+ str(int(acc*100))+'%')

            def draw_lines(self, event):
                self.x = event.x
                self.y = event.y
                r=8
                self.canvas.create_oval(self.x-r, self.y-r, self.x + r, self.y + r, fill='black')
        app = App()
        mainloop()
```

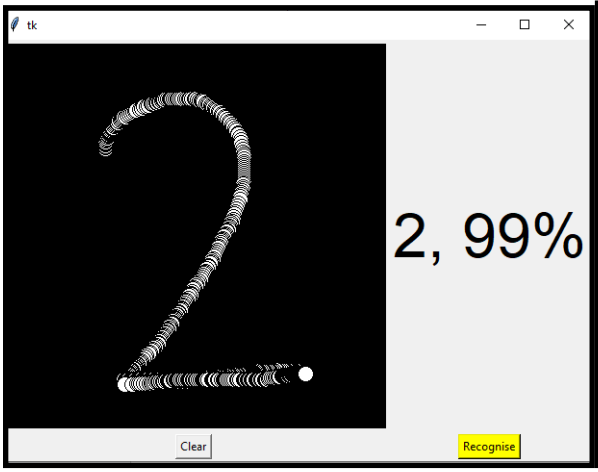
To build an interactive window we have created a new file in GUI. In this file, we can draw digits on canvas, and by clicking a button, we can identify the digit. The Tkinter library is part of the Python standard library.

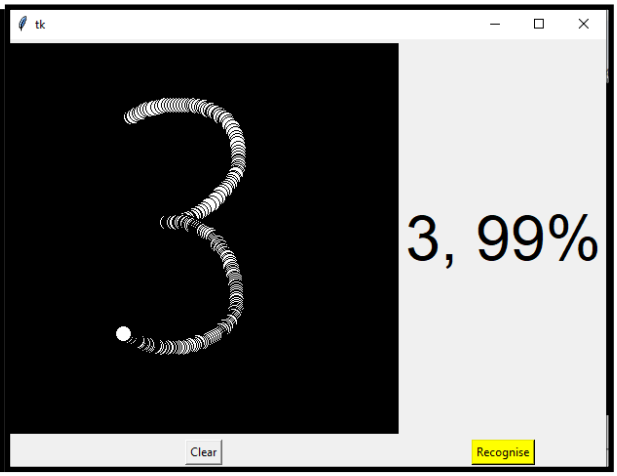
`predict_digit()` method takes the picture as input and then activates the trained model to predict the digit. After that to build the GUI for our app we have created the App class. In GUI canvas we can draw a digit by capturing the mouse event and with a button click, we hit the **`predict_digit()`** function and show the results.

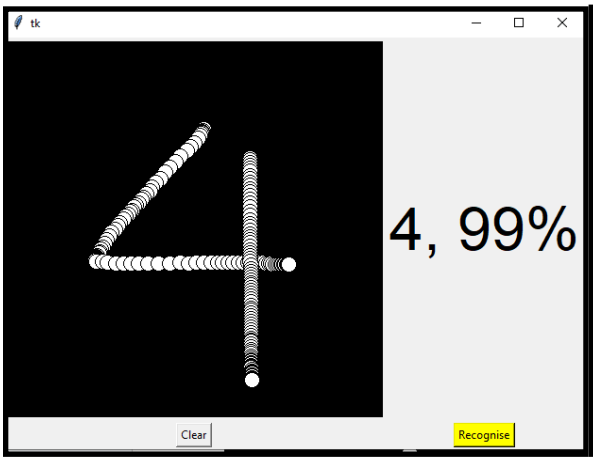
RESULTS OF PREDICTION

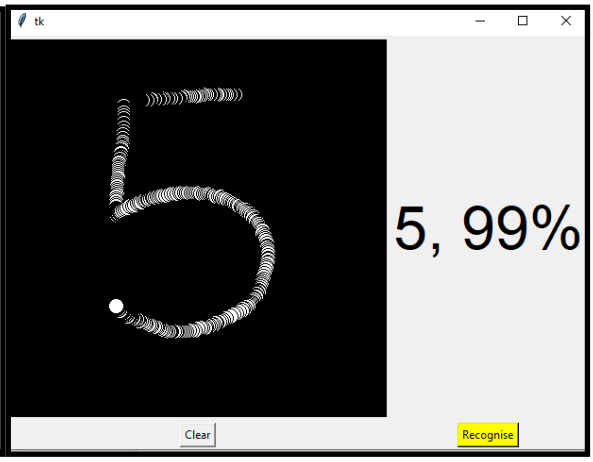












Conclusion

In this report, we begin by discussing the Convolutional Neural Network and its importance. Then we explained about the dataset & how a dataset is divided into training and test dataset. The dataset was cleaned, scaled, and shaped. Using TensorFlow, a CNN model was created and was eventually trained on the training dataset. Finally, predictions were made using the trained model.

We built the GUI for easy learning where we drew a digit on the canvas then we classified the digit and showed the results.