**Federated learning implementation in mnist handwritten data**

Presented by,

B.Rahul

CONTENT

* Introduction
* Import dataset
* Prediction and Plot
* Global loss and Accuracy
* Comparison between with gpu and without gpu implementation
* Conclusion

Introduction

MNIST is a widely used dataset of handwritten digits that contains 60,000 handwritten digits for training a machine learning model and 10,000 handwritten digits for testing the model. It was introduced in 1998 and has become a standard benchmark for classification tasks. It is also called the “Hello, World” dataset as it’s very easy to use. MNIST was derived from an even larger dataset, the NIST special database which not only contains digits but also uppercase and lowercase handwritten letters.

IMPORT DATASET

The dataset was directly imported from google drive by mounting the files.

Syntax:

from google.colab import drive

drive.mount('/content/drive')

!unzip /content/drive/MyDrive/federated.zip

Output:

**Streaming output truncated to the last 5000 lines.**

inflating: trainingSet/trainingSet/8/img\_4000.jpg

inflating: trainingSet/trainingSet/8/img\_40004.jpg

inflating: trainingSet/trainingSet/8/img\_4001.jpg

inflating: trainingSet/trainingSet/8/img\_40019.jpg

inflating: trainingSet/trainingSet/8/img\_40026.jpg

inflating: trainingSet/trainingSet/8/img\_40029.jpg

inflating: trainingSet/trainingSet/8/img\_40032.jpg

inflating: trainingSet/trainingSet/8/img\_40055.jpg

TRAINING AND TESTING THE MODEL

Packages imported:

!pip install --quiet --upgrade tensorflow-federated

!pip install --quiet --upgrade nest-asyncio

import collections

import numpy as np

import tensorflow as tf

import tensorflow\_federated as tff

Syntax:

emnist\_train, emnist\_test = tff.simulation.datasets.emnist.load\_data()

Length of client ids:

len(emnist\_train.client\_ids)

output: 3383

Syntax for plot:

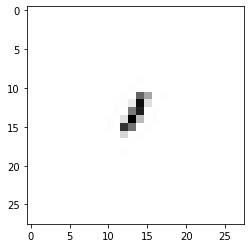
from matplotlib import pyplot as plt

plt.imshow(example\_element['pixels'].numpy(), cmap='gray', aspect='equal')

plt.grid(False)

\_ = plt.show()

Output plot:



Syntax for mnist dataset for one client:

Now let's visualize the number of examples on each client for each MNIST digit label. In the federated environment, the number of examples on each client can vary quite a bit, depending on user behavior.

## Example MNIST digits for one client

figure = plt.figure(figsize=(20, 4))

j = 0

for example in example\_dataset.take(40):

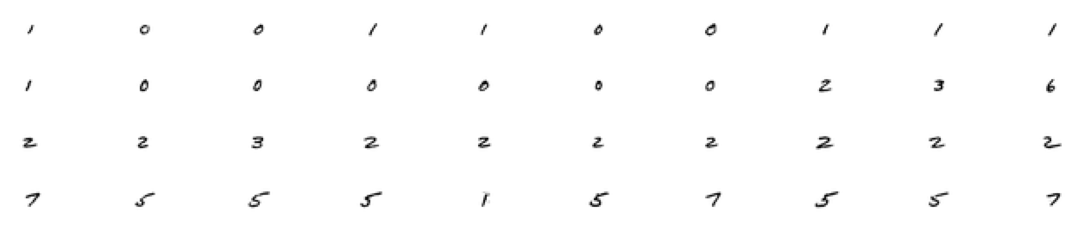
  plt.subplot(4, 10, j+1)

  plt.imshow(example['pixels'].numpy(), cmap='gray', aspect='equal')

  plt.axis('off')

  j += 1

Output:



Syntax for label count for sample clients:

# Number of examples per layer for a sample of clients

f = plt.figure(figsize=(12, 7))

f.suptitle('Label Counts for a Sample of Clients')

for i in range(6):

  client\_dataset = emnist\_train.create\_tf\_dataset\_for\_client(

      emnist\_train.client\_ids[i])

  plot\_data = collections.defaultdict(list)

  for example in client\_dataset:

    # Append counts individually per label to make plots

    # more colorful instead of one color per plot.

    label = example['label'].numpy()

    plot\_data[label].append(label)

  plt.subplot(2, 3, i+1)

  plt.title('Client {}'.format(i))

  for j in range(10):

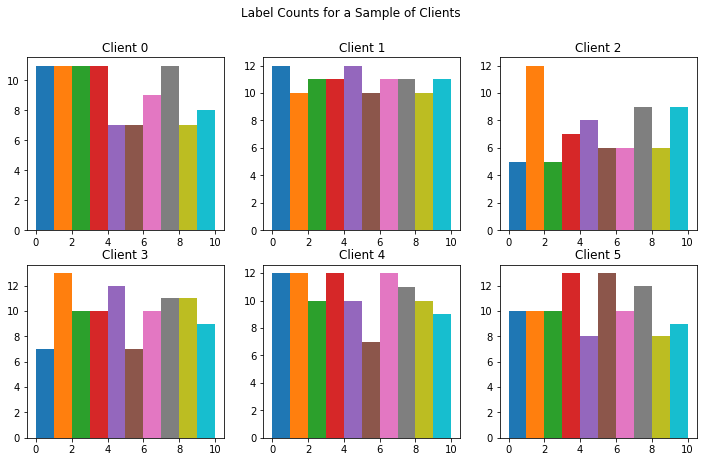
    plt.hist(

        plot\_data[j],

        density=False,

        bins=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

Output:



Syntax for mean of client images per label:

# Each client has different mean images, meaning each client will be nudging

# the model in their own directions locally.

for i in range(5):

  client\_dataset = emnist\_train.create\_tf\_dataset\_for\_client(

      emnist\_train.client\_ids[i])

  plot\_data = collections.defaultdict(list)

  for example in client\_dataset:

    plot\_data[example['label'].numpy()].append(example['pixels'].numpy())

  f = plt.figure(i, figsize=(12, 5))

  f.suptitle("Client #{}'s Mean Image Per Label".format(i))

  for j in range(10):

    mean\_img = np.mean(plot\_data[j], 0)

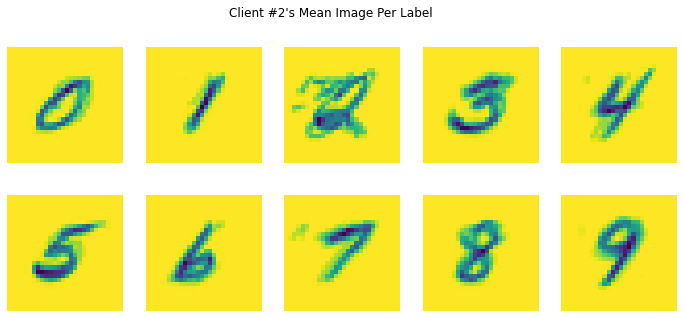
    plt.subplot(2, 5, j+1)

    plt.imshow(mean\_img.reshape((28, 28)))

    plt.axis('off')

Output:





DATA PREPROCESSING:

Syntax:

NUM\_CLIENTS = 10

NUM\_EPOCHS = 5

BATCH\_SIZE = 20

SHUFFLE\_BUFFER = 100

PREFETCH\_BUFFER = 10

def preprocess(dataset):

  def batch\_format\_fn(element):

    """Flatten a batch `pixels` and return the features as an `OrderedDict`."""

    return collections.OrderedDict(

        x=tf.reshape(element['pixels'], [-1, 784]),

        y=tf.reshape(element['label'], [-1, 1]))

  return dataset.repeat(NUM\_EPOCHS).shuffle(SHUFFLE\_BUFFER, seed=1).batch(

      BATCH\_SIZE).map(batch\_format\_fn).prefetch(PREFETCH\_BUFFER)

Syntax for finding number of client datasets:

sample\_clients = emnist\_train.client\_ids[0:NUM\_CLIENTS]

federated\_train\_data = make\_federated\_data(emnist\_train, sample\_clients)

print(f'Number of client datasets: {len(federated\_train\_data)}')

print(f'First dataset: {federated\_train\_data[0]}')

Output:

Number of client datasets: 10

First dataset: <\_PrefetchDataset element\_spec=OrderedDict([('x', TensorSpec(shape=(None, 784), dtype=tf.float32, name=None)), ('y', TensorSpec(shape=(None, 1), dtype=tf.int32, name=None))])>

PREDICION

Syntax:

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

fig = plt.figure()

for i in range(9):

  plt.subplot(3,3,i+1)

  plt.tight\_layout()

  plt.imshow(X\_train[i], cmap='gray', interpolation='none')

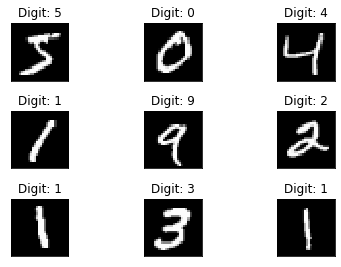
  plt.title("Digit: {}".format(y\_train[i]))

  plt.xticks([])

  plt.yticks([])

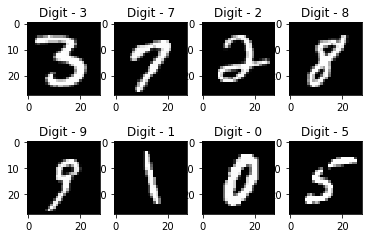
fig

Output:



Prediction results:

Handwritten data recognising:



Syntax for pixel estimation:

fig = plt.figure()

plt.subplot(2,1,1)

plt.imshow(X\_train[0], cmap='gray', interpolation='none')

plt.title("Digit: {}".format(y\_train[0]))

plt.xticks([])

plt.yticks([])

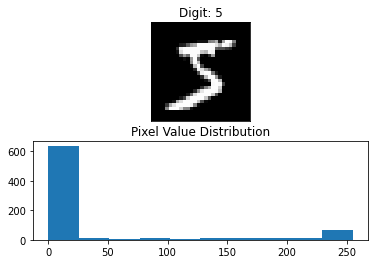
plt.subplot(2,1,2)

plt.hist(X\_train[0].reshape(784))

plt.title("Pixel Value Distribution")

fig

Output:



OUTPUT METRICS - ACCURACY

Syntax for building the global and loss accuracy:

# compiling the sequential model

model.compile(loss='categorical\_crossentropy', metrics=['accuracy'], optimizer='adam')

# training the model and saving metrics in history

history = model.fit(X\_train, Y\_train,

          batch\_size=128, epochs=20,

          verbose=2,

          validation\_data=(X\_test, Y\_test))

# saving the model

save\_dir = "/results/"

model\_name = 'keras\_mnist.h5'

model\_path = os.path.join(save\_dir, model\_name)

model.save(model\_path)

print('Saved trained model at %s ' % model\_path)

# plotting the metrics

fig = plt.figure()

plt.subplot(2,1,1)

#plt.plot(history.history['acc'])

#plt.plot(history.history['val\_acc'])

plt.title('model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='lower right')

plt.subplot(2,1,2)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('model loss')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper right')

plt.tight\_layout()

fig

Output:

Epoch 1/20

469/469 - 11s - loss: 0.0136 - accuracy: 0.9955 - val\_loss: 0.0851 - val\_accuracy: 0.9830 - 11s/epoch - 23ms/step

Epoch 2/20

469/469 - 9s - loss: 0.0159 - accuracy: 0.9950 - val\_loss: 0.0887 - val\_accuracy: 0.9817 - 9s/epoch - 19ms/step

Epoch 3/20

469/469 - 10s - loss: 0.0164 - accuracy: 0.9951 - val\_loss: 0.0858 - val\_accuracy: 0.9849 - 10s/epoch - 22ms/step

Epoch 4/20

469/469 - 10s - loss: 0.0124 - accuracy: 0.9959 - val\_loss: 0.0917 - val\_accuracy: 0.9838 - 10s/epoch - 21ms/step

Epoch 5/20

469/469 - 10s - loss: 0.0129 - accuracy: 0.9956 - val\_loss: 0.0865 - val\_accuracy: 0.9847 - 10s/epoch - 21ms/step

Epoch 6/20

469/469 - 9s - loss: 0.0127 - accuracy: 0.9959 - val\_loss: 0.0825 - val\_accuracy: 0.9839 - 9s/epoch - 18ms/step

Epoch 7/20

469/469 - 10s - loss: 0.0109 - accuracy: 0.9966 - val\_loss: 0.1017 - val\_accuracy: 0.9826 - 10s/epoch - 21ms/step

Epoch 8/20

469/469 - 10s - loss: 0.0152 - accuracy: 0.9955 - val\_loss: 0.0956 - val\_accuracy: 0.9831 - 10s/epoch - 22ms/step

Epoch 9/20

469/469 - 10s - loss: 0.0095 - accuracy: 0.9970 - val\_loss: 0.0820 - val\_accuracy: 0.9850 - 10s/epoch - 21ms/step

Epoch 10/20

469/469 - 9s - loss: 0.0127 - accuracy: 0.9959 - val\_loss: 0.0898 - val\_accuracy: 0.9826 - 9s/epoch - 19ms/step

Epoch 11/20

469/469 - 11s - loss: 0.0116 - accuracy: 0.9959 - val\_loss: 0.0784 - val\_accuracy: 0.9842 - 11s/epoch - 24ms/step

Epoch 12/20

469/469 - 10s - loss: 0.0126 - accuracy: 0.9962 - val\_loss: 0.0792 - val\_accuracy: 0.9848 - 10s/epoch - 22ms/step

Epoch 13/20

469/469 - 10s - loss: 0.0089 - accuracy: 0.9973 - val\_loss: 0.0970 - val\_accuracy: 0.9834 - 10s/epoch - 22ms/step

Epoch 14/20

469/469 - 9s - loss: 0.0126 - accuracy: 0.9960 - val\_loss: 0.0858 - val\_accuracy: 0.9845 - 9s/epoch - 19ms/step

Epoch 15/20

469/469 - 10s - loss: 0.0106 - accuracy: 0.9966 - val\_loss: 0.1016 - val\_accuracy: 0.9836 - 10s/epoch - 21ms/step

Epoch 16/20

469/469 - 11s - loss: 0.0132 - accuracy: 0.9960 - val\_loss: 0.0892 - val\_accuracy: 0.9828 - 11s/epoch - 24ms/step

Epoch 17/20

469/469 - 10s - loss: 0.0103 - accuracy: 0.9966 - val\_loss: 0.0816 - val\_accuracy: 0.9850 - 10s/epoch - 21ms/step

Epoch 18/20

469/469 - 9s - loss: 0.0104 - accuracy: 0.9965 - val\_loss: 0.0973 - val\_accuracy: 0.9840 - 9s/epoch - 20ms/step

Epoch 19/20

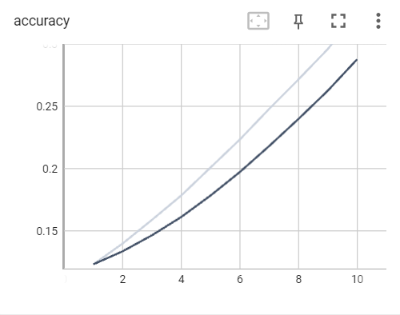
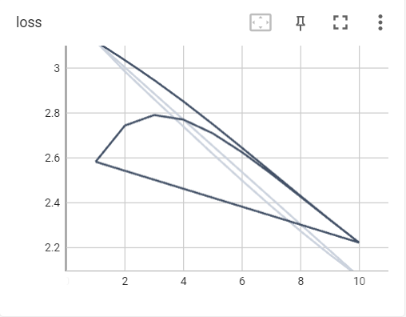
469/469 - 9s - loss: 0.0091 - accuracy: 0.9972 - val\_loss: 0.1024 - val\_accuracy: 0.9831 - 9s/epoch - 19ms/step

Epoch 20/20

469/469 - 10s - loss: 0.0107 - accuracy: 0.9971 - val\_loss: 0.0965 - val\_accuracy: 0.9841 - 10s/epoch - 21ms/step

Saved trained model at /results/keras\_mnist.h5

Plot for the model accuracy and loss using tensor flow board:

Syntax for final measure metrics:

mnist\_model = load\_model('keras\_mnist.h5')

loss\_and\_metrics = mnist\_model.evaluate(X\_test, Y\_test, verbose=2)

print("Test Loss", loss\_and\_metrics[0])

print("Test Accuracy", loss\_and\_metrics[1])

Output:

313/313 - 2s - loss: 0.0745 - accuracy: 0.9822 - 2s/epoch - 6ms/step

Test Loss 0.07445215433835983

Test Accuracy 0.982200026512146

GPU IMPLEMENTATION:

The gpu implementation was done by changing the runtime type setting of colab to gpu accelator.

Package imported:

!pip install numba

!pip install cuda-python

!pip install xlwings

!pip install tensorflow-gpu

Syntax for load,preprocess,train and evalvuate model:

# Import necessary libraries

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import numpy as np

# Load the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

# Preprocess the data

x\_train = x\_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

# Define the model

model = keras.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test), verbose=2)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print('Test accuracy:', test\_acc)

Output:

Epoch 1/10

1875/1875 - 7s - loss: 0.2237 - accuracy: 0.9380 - val\_loss: 0.0870 - val\_accuracy: 0.9746 - 7s/epoch - 3ms/step

Epoch 2/10

1875/1875 - 5s - loss: 0.0838 - accuracy: 0.9758 - val\_loss: 0.0644 - val\_accuracy: 0.9783 - 5s/epoch - 3ms/step

Epoch 3/10

1875/1875 - 6s - loss: 0.0616 - accuracy: 0.9821 - val\_loss: 0.0642 - val\_accuracy: 0.9801 - 6s/epoch - 3ms/step

Epoch 4/10

1875/1875 - 5s - loss: 0.0519 - accuracy: 0.9842 - val\_loss: 0.0553 - val\_accuracy: 0.9828 - 5s/epoch - 3ms/step

Epoch 5/10

1875/1875 - 5s - loss: 0.0437 - accuracy: 0.9870 - val\_loss: 0.0525 - val\_accuracy: 0.9836 - 5s/epoch - 3ms/step

Epoch 6/10

1875/1875 - 6s - loss: 0.0377 - accuracy: 0.9883 - val\_loss: 0.0538 - val\_accuracy: 0.9833 - 6s/epoch - 3ms/step

Epoch 7/10

1875/1875 - 5s - loss: 0.0320 - accuracy: 0.9901 - val\_loss: 0.0543 - val\_accuracy: 0.9833 - 5s/epoch - 2ms/step

Epoch 8/10

1875/1875 - 6s - loss: 0.0280 - accuracy: 0.9918 - val\_loss: 0.0543 - val\_accuracy: 0.9835 - 6s/epoch - 3ms/step

Epoch 9/10

1875/1875 - 5s - loss: 0.0237 - accuracy: 0.9928 - val\_loss: 0.0559 - val\_accuracy: 0.9845 - 5s/epoch - 3ms/step

Epoch 10/10

1875/1875 - 5s - loss: 0.0205 - accuracy: 0.9935 - val\_loss: 0.0600 - val\_accuracy: 0.9828 - 5s/epoch - 2ms/step

313/313 - 1s - loss: 0.0600 - accuracy: 0.9828 - 915ms/epoch - 3ms/step

Test accuracy: 0.9828000068664551

Syntax for implemtation of gpu with tensorflow :

model = tf.keras.Sequential([

  tf.keras.layers.Flatten(input\_shape=(28, 28)),

  tf.keras.layers.Dense(128, activation='relu'),

  tf.keras.layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

history = model.fit(x\_train, y\_train,

                    epochs=10,

                    validation\_data=(x\_test, y\_test))

Output:

Epoch 1/10

1875/1875 [==============================] - 12s 4ms/step - loss: 0.2596 - accuracy: 0.9252 - val\_loss: 0.1491 - val\_accuracy: 0.9535

Epoch 2/10

1875/1875 [==============================] - 9s 5ms/step - loss: 0.1165 - accuracy: 0.9652 - val\_loss: 0.0992 - val\_accuracy: 0.9698

Epoch 3/10

1875/1875 [==============================] - 8s 4ms/step - loss: 0.0790 - accuracy: 0.9759 - val\_loss: 0.0930 - val\_accuracy: 0.9717

Epoch 4/10

1875/1875 [==============================] - 7s 4ms/step - loss: 0.0590 - accuracy: 0.9823 - val\_loss: 0.0778 - val\_accuracy: 0.9769

Epoch 5/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0452 - accuracy: 0.9861 - val\_loss: 0.0789 - val\_accuracy: 0.9765

Epoch 6/10

1875/1875 [==============================] - 7s 4ms/step - loss: 0.0359 - accuracy: 0.9890 - val\_loss: 0.0859 - val\_accuracy: 0.9741

Epoch 7/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0293 - accuracy: 0.9912 - val\_loss: 0.0895 - val\_accuracy: 0.9754

Epoch 8/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0240 - accuracy: 0.9927 - val\_loss: 0.0822 - val\_accuracy: 0.9763

Epoch 9/10

1875/1875 [==============================] - 6s 3ms/step - loss: 0.0186 - accuracy: 0.9942 - val\_loss: 0.0776 - val\_accuracy: 0.9792

Epoch 10/10

1875/1875 [==============================] - 5s 3ms/step - loss: 0.0166 - accuracy: 0.9949 - val\_loss: 0.0897 - val\_accuracy: 0.9768

COMPARISON BETWEEN WITH GPU AND WITHOUT GPU IMPLEMNTATION:

SYNTAX:

from numba import jit, cuda

import numpy as np

# to measure exec time

from timeit import default\_timer as timer

# normal function to run on cpu

def func(a):

  for i in range(10000000):

    a[i]+= 1

# function optimized to run on gpu

@jit(target\_backend='cuda')

def func2(a):

  for i in range(10000000):

    a[i]+= 1

if \_\_name\_\_=="\_\_main\_\_":

  n = 10000000

  a = np.ones(n, dtype = np.float64)

  start = timer()

  func(a)

  print("without GPU:", timer()-start)

  start = timer()

  func2(a)

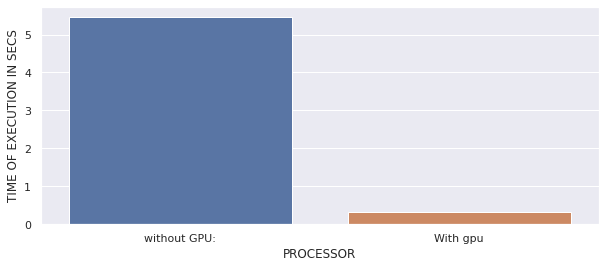
  print("with GPU:", timer()-start)

Output:

without GPU: 5.456030191000025

with GPU: 0.31363791500001525

COMPARISON PLOT FOR WITHOUT AND WITH GPU:



HANDWRITTEN DATA RECOGNISATION:

GUI based prediction results:

SYNTAX:

from keras.models import load\_model

from tkinter import \*

import tkinter as tk

import win32gui

from PIL import ImageGrab, Image

import numpy as np

model = load\_model('mnist.h5')

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 input and normalizing

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)

class App(tk.Tk):

def \_\_init\_\_(self):

tk.Tk.\_\_init\_\_(self)

self.x = self.y = 0

# Creating elements

self.canvas = tk.Canvas(self, width=300, height=300, bg = "white", cursor="cross")

self.label = tk.Label(self, text="Thinking..", font=("Helvetica", 48))

self.classify\_btn = tk.Button(self, text = "Recognise", command = self.classify\_handwriting)

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

rect = win32gui.GetWindowRect(HWND) # get the coordinate of the canvas

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

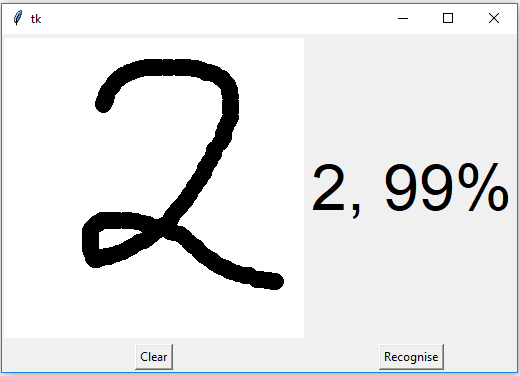
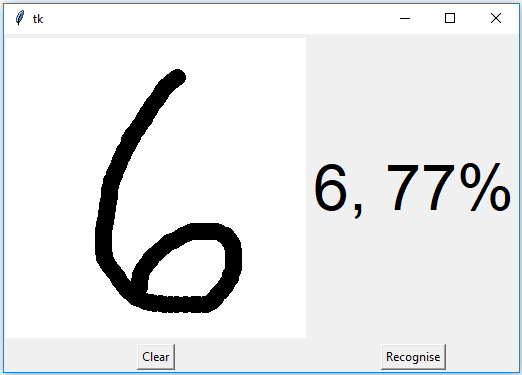
r=8

self.canvas.create\_oval(self.x-r, self.y-r, self.x + r, self.y + r, fill='black')

app = App()

mainloop()

OUTPUT:

CONCLUSION

The model have been trained with use of gpu that yield a good results with speed execution.

GPU’s have more cores than CPU and hence when it comes to parallel computing of data, GPUs perform exceptionally better than CPUs even though GPUs has lower clock speed and it lacks several core management features as compared to the CPU.

Thus, running a python script on GPU can prove to be comparatively faster than CPU, however, it must be noted that for processing a data set with GPU, the data will first be transferred to the GPU’s memory which may require additional time so if data set is small then CPU may perform better than GPU.

THANK YOU!!!