SANT DNYANESHWAR SHIKSHAN SANSTHA'S
ANNASAHEB DANGE COLLEGE OF ENGINEERING & TECHNOLOGY, ASHTA

DEPARTMENT OF ARTIFICIAL INTELLIENCE AND DATA SCIENCE

LABORATORY MANUAL

Deep Learning

Mr. Vikas Honmane

THIRD YEAR B.TECH

Academic Year

2024-25

Experiments List

Experiments :	
1.	Designing and developing a model for Autoencoder for Dimensionality reduction
2.	Implement character and Digit Recognition using ANN.
3.	Implement the analysis of X-ray images using autoencoders
4.	Implement Speech Recognition using NLP, For Home automation
5.	Develop a code to design object detection and classification for traffic analysis using CNN
6.	Designing and developing a model for Text generation using LSTM
7.	Implement image augmentation using deep RBM.
8.	Designing and developing a model for Autoencoder for classification
9.	Micro Project: Number plate recognition of traffic video analysis.

DOs and DON'Ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.
2. All the students should sit according to their roll numbers starting from their left to right.
3. All the students are supposed to enter the Presence in the log book.
4. Do not change the Computer on which you are working.
5. All the students are expected to get at least the algorithm of the program/concept to be implemented.
6. Strictly observe the instructions given by the teacher/Lab Instructor.
7. Do not disturb machine Hardware / Software Setup.

Instruction for Laboratory Teachers

1. Submission related to whatever lab work has been completed should be done during the next lab session along with signing the index.

2. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.

3. Continuous assessment in the prescribed format must be followed.

Expt. No. 1	Designing and developing a model for Autoencoder for Dimensionality reduction
Date :	

Aim: To study and demonstrate basics of Autoencoder for Dimensionality reduction on MNIST dataset

Purpose:

 Reduce the number of features in a dataset while retaining its significant structures and patterns.

Objectives:

1. Explain working of Autoencoders.
2. Design autoencoder for dimensionality reduction on MNIST dataset.
3. Implement denoising autoencoder.

Equipment's /Components / Software:

S.No	Name
1	Google colab

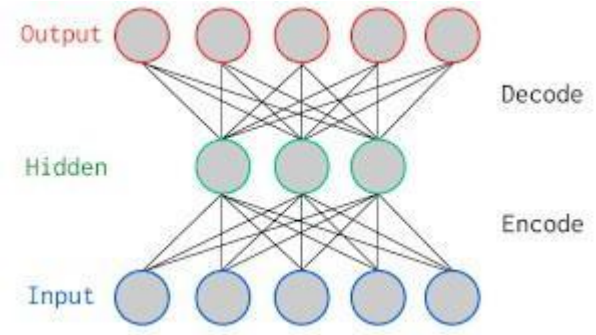
Theory:

Autoencoder

an autoencoder is an unsupervised machine learning algorithm that takes an image as input and tries to reconstruct it using fewer number of bits from the bottleneck also known as latent space. The image is majorly compressed at the bottleneck. The compression in autoencoders is achieved by training the network for a period of time and as it learns it tries to best represent the input image at the bottleneck. The general image compression algorithms like JPEG and JPEG lossless compression techniques compress the images without the need for any kind of training and do fairly well in compressing the images.

Autoencoders are similar to dimensionality reduction techniques like Principal Component Analysis (PCA). They project the data from a higher dimension to a lower dimension using linear transformation and try to preserve the important features of the data while removing the non-essential parts.

However, the major difference between autoencoders and PCA lies in the transformation part: as you already read, PCA uses linear transformation whereas autoencoders use non-linear transformations.

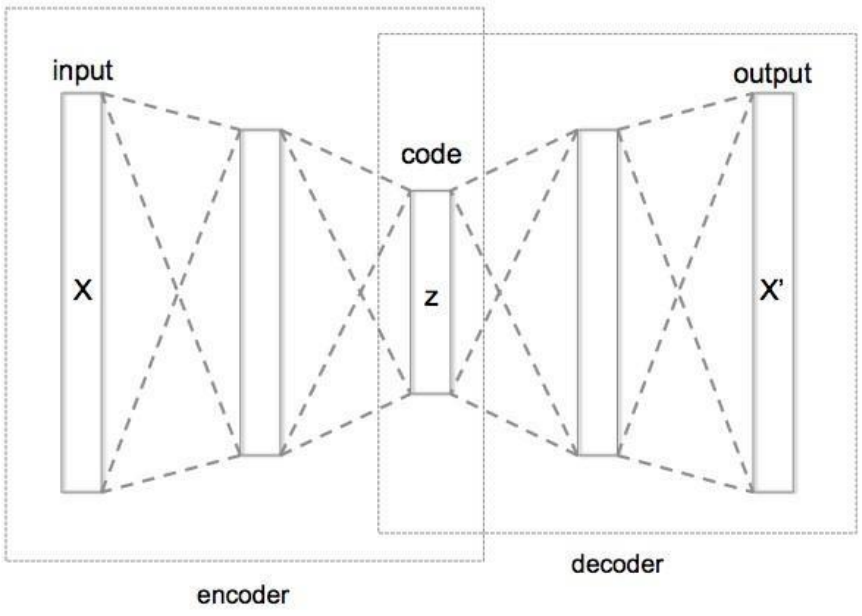


The above figure is a two-layer vanilla autoencoder with one hidden layer. In deep learning terminology, you will often notice that the input layer is never taken into account while counting the total number of layers in an architecture. The total layers in an architecture only comprises of the number of hidden layers and the ouput layer.

As shown in the image above, the input and output layers have the same number of neurons.

Let's take an example. You feed an image with just five pixel values into the autoencoder which is compressed by the encoder into three pixel values at the bottleneck (middle layer) or latent space. Using these three values, the decoder tries to reconstruct the five pixel values or rather the input image which you fed as an input to the network.

In reality, there are more number of hidden layers in between the input and the output.



Autoencoder can be broken in to three parts

- Encoder: this part of the network compresses or downsamples the input into a fewer number of bits. The space represented by these fewer number of bits is often called the *latentspace* or *bottleneck*. The bottleneck is also called the "maximum point of compression" since at this point the input is compressed the maximum. These compressed bits that represent the

  original input are together called an “encoding” of the input.

- Decoder: this part of the network tries to reconstruct the input using only the encoding of the input. When the decoder is able to reconstruct the input exactly as it was fed to the encoder, you can say that the encoder is able to produce the best encodings for the input with which the decoder is able to reconstruct well!

There are variety of autoencoders, such as the convolutional autoencoder, denoising autoencoder, variational autoencoder and sparse autoencoder. However, as you read in the introduction, you'll only focus on the convolutional and denoising ones in this tutorial

Code and Result:

```
# import libraries
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F

import matplotlib.pyplot as plt
import matplotlib_inline.backend_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
```

Import and process the data

```
# import dataset (comes with colab!)
data = np.loadtxt(open('sample_data/mnist_train_small.csv','rb'),delimiter=',')

# don't need labels!
data = data[:,1:]

# normalize the data to a range of [0 1]
dataNorm = data / np.max(data)

# convert to tensor
```

```
dataT = torch.tensor( dataNorm ).float()
```

Create the DL model

```
# create a class for the model
def createTheMNISTAE():

  class aenet(nn.Module):
    def __init__(self):
      super().__init__()

      ### input layer
      self.input = nn.Linear(784,250)

      ### encoder layer
      self.enc = nn.Linear(250,50)

      ### latent layer
      self.lat = nn.Linear(50,250)

      ### decoder layer
      self.dec = nn.Linear(250,784)

    # forward pass
    def forward(self,x):
      x = F.relu( self.input(x) )
      x = F.relu( self.enc(x) )
      x = F.relu( self.lat(x) )
      y = torch.sigmoid( self.dec(x) )
      return y

  # create the model instance
  net = aenet()

  # loss function
  lossfun = nn.MSELoss()

  # optimizer
  optimizer = torch.optim.Adam(net.parameters(),lr=.001)

  return net,lossfun,optimizer
# test the model with a bit of data
net,lossfun,optimizer = createTheMNISTAE()

X = dataT[5:10,:]
yHat = net(X)

print(X.shape)
print(yHat.shape)
# let's see what the model did!
```

```
fig,axs = plt.subplots(2,5,figsize=(10,3))

for i in range(5):
  axs[0,i].imshow(X[i,:].view(28,28).detach() ,cmap='gray')
  axs[1,i].imshow(yHat[i,:].view(28,28).detach() ,cmap='gray')
  axs[0,i].set_xticks([]), axs[0,i].set_yticks([])
  axs[1,i].set_xticks([]), axs[1,i].set_yticks([])

plt.suptitle('Yikes!!!')
plt.show()
```

Create a function that trains the model

```
def function2trainTheModel():

  # number of epochs
  numepochs = 10000

  # create a new model
  net,lossfun,optimizer = createTheMNISTAE()

  # initialize losses
  losses = torch.zeros(numepochs)

  # loop over epochs
  for epochi in range(numepochs):

    # select a random set of images
    randomidx = np.random.choice(dataT.shape[0],size=32)
    X = dataT[randomidx,:]

    # forward pass and loss
    yHat = net(X)
    loss = lossfun(yHat,X)

    # backprop
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # losses in this epoch
    losses[epochi] = loss.item()
  # end epochs

  # function output
  return losses,net
```

Run the model and show the results!

```
# train the model (90s)
losses,net = function2trainTheModel()
print(f'Final loss: {losses[-1]:.4f}')

# visualize the losses
```

```
plt.plot(losses,'.-')
plt.xlabel('Epochs')
plt.ylabel('Model loss')
plt.title('OK, but what did it actually learn??')
plt.show()
# Repeat the visualization when testing the model
X = dataT[:5,:]
yHat = net(X)

# let's see what the model did!
fig,axs = plt.subplots(2,5,figsize=(10,3))

for i in range(5):
  axs[0,i].imshow(X[i,:].view(28,28).detach() ,cmap='gray')
  axs[1,i].imshow(yHat[i,:].view(28,28).detach() ,cmap='gray')
  axs[0,i].set_xticks([]), axs[0,i].set_yticks([])
  axs[1,i].set_xticks([]), axs[1,i].set_yticks([])

plt.suptitle('Disregard the yikes!!!')
plt.show()
Add noise to see a use case of an autoencoder
# grab a small set of images
X = dataT[:10,:]

# add noise
Xnoise = X + torch.rand_like(X)/4

# clip at 1
Xnoise[Xnoise>1] = 1

# show the noisy images
fig,axs = plt.subplots(2,5,figsize=(10,3))

for i in range(5):
  axs[0,i].imshow(X[i,:].view(28,28).detach() ,cmap='gray')
  axs[1,i].imshow(Xnoise[i,:].view(28,28).detach() ,cmap='gray')
  axs[0,i].set_xticks([]), axs[0,i].set_yticks([])
  axs[1,i].set_xticks([]), axs[1,i].set_yticks([])

plt.show()

# show the noisy images
fig,axs = plt.subplots(3,10,figsize=(12,5))

for i in range(10):
  axs[0,i].imshow(X[i,:].view(28,28).detach() ,cmap='gray')
  axs[1,i].imshow(Xnoise[i,:].view(28,28).detach() ,cmap='gray')
  axs[2,i].imshow(Y[i,:].view(28,28).detach() ,cmap='gray')
  axs[0,i].set_xticks([]), axs[0,i].set_yticks([])
  axs[1,i].set_xticks([]), axs[1,i].set_yticks([])
```

```
  axs[2,i].set_xticks([]), axs[2,i].set_yticks([])

plt.suptitle('Neato.')
plt.show()
```

Conclusion:

The autoencoder successfully captured the underlying structure of the input data, enabling effective reconstruction with minimal loss.

Sr. No.	Question	CO Mapped
1	What are the primary components of an autoencoder, and how do they contribute to the model's functionality?	1ADPC314_1
2	How does an autoencoder differ from other dimensionality reduction techniques such as PCA (Principal Component Analysis)?	1ADPC314_4
3	What are some common applications of autoencoders in various fields, and how do they benefit from the use of autoencoders?	1ADPC314_4
4	How do different types of autoencoders, such as variational autoencoders (VAEs) and convolutional autoencoders, enhance the capabilities of standard autoencoders?	1ADPC314_4
5	What are the main challenges associated with training autoencoders, and how can they be addressed to improve model performance and generalization?	1ADPC314_4

Student Activity:

Implement the code, write theory ,come with Printout of that	

Other Links:

https://www.geeksforgeeks.org/implementing-an-autoencoder-in-pytorch/

Expt. No. 2	Implement character and digit recognition using ANN.
Date :	

Aim: To study and demonstrate handwritten digit recognition using MNIST dataset using neural network along with an integrated GUI

Purpose:

 Recognize handwritten digits in a dataset with the help of Artificial Neural Network.

Objectives:

1. Explain working of Artificial Neural Network.
2. Illustrate the usage of Module in python by making user defined library and import user defined library.
3. Design User interface using Tkinter library.
4. Implement Artificial Neural Network on MNIST dataset to recognize handwritten digits on canvas.

Equipment's /Components / Software:

S.No	Name
1	Pycharm or VScode

Theory:

Introduction:

Handwritten digit recognition using MNIST dataset is a major project made with the help of Neural Network. It basically detects the scanned images of handwritten digits.

We have taken this a step further where our handwritten digit recognition system not only detects scanned images of handwritten digits but also allows writing digits on the screen with the help of an integrated GUI for recognition.

Approach:

We will approach this project by using a three-layered Neural Network.

The input layer: It distributes the features of our examples to the next layer for calculation of activations of the next layer.

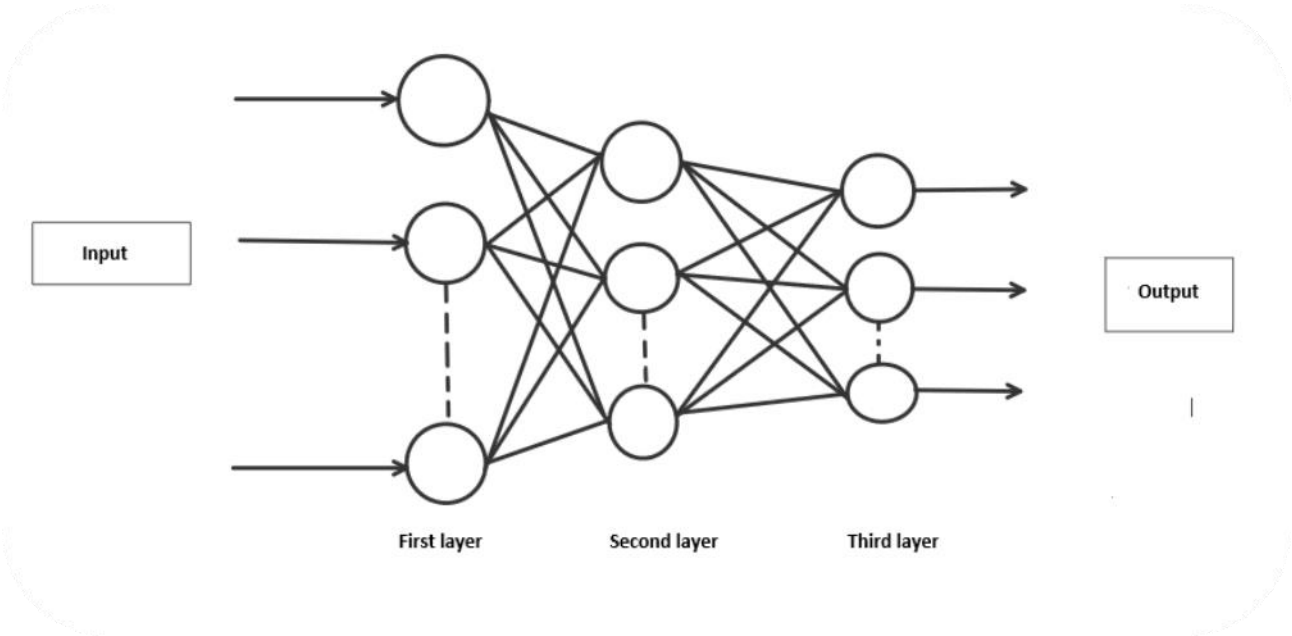
The hidden layer: They are made of hidden units called activations providing nonlinear ties for the network. A number of hidden layers can vary according to our requirements.

The output layer: The nodes here are called output units. It provides us with the final prediction of the Neural Network on the basis of which final predictions can be made.

A neural network is a model inspired by how the brain works. It consists of multiple layers having many activations, this activation resembles neurons of our brain. A neural network tries to learn a set of parameters in a set of data which could help to recognize the underlying relationships. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria.

Methodology:

We have implemented a Neural Network with 1 hidden layer having 100 activation units (excluding bias units). The data is loaded from a .mat file, features(X) and labels(y) were extracted. Then features are divided by 255 to rescale them into a range of [0,1] to avoid overflow during computation. Data is split up into 60,000 training and 10,000 testing examples. Feedforward is performed with the training set for calculating the hypothesis and then backpropagation is done in order to reduce the error between the layers. The regularization parameter lambda is set to 0.1 to address the problem of overfitting. Optimizer is run for 70 iterations to find the best fit model.



Coding:

Main.py

Importing all the required libraries, extract the data from mnist-original.mat file. Then features and labels will be separated from extracted data. After that data will be split into training (60,000) and testing (10,000) examples. Randomly initialize Thetas in the range of [-0.15, +0.15] to break symmetry and get better results. Further, the optimizer is called for the training of weights, to minimize the cost function for appropriate predictions. We have used the “minimize” optimizer from “scipy.optimize” library with “L-BFGS-B” method. We have calculated the test, the “training set accuracy and precision using “predict” function.

```
from scipy.io import loadmat
import numpy as np
from Model import neural_network
from RandInitialize import initialise
from Prediction import predict
from scipy.optimize import minimize



# Loading mat file
data = loadmat('mnist-original.mat')

# Extracting features from mat file
X = data['data']
X = X.transpose()

# Normalizing the data
X = X / 255

# Extracting labels from mat file
y = data['label']
y = y.flatten()
```

```
# Splitting data into training set with 60,000 examples
X_train = X[:60000, :]
y_train = y[:60000]

# Splitting data into testing set with 10,000 examples
X_test = X[60000:, :]
y_test = y[60000:]

m = X.shape[0]
input_layer_size = 784 # Images are of (28 X 28) px so there will be 784 features
hidden_layer_size = 100
num_labels = 10 # There are 10 classes [0, 9]

# Randomly initialising Thetas
initial_Theta1 = initialise(hidden_layer_size, input_layer_size)
initial_Theta2 = initialise(num_labels, hidden_layer_size)

# Unrolling parameters into a single column vector
initial_nn_params = np.concatenate((initial_Theta1.flatten(), initial_Theta2.flatten()))
maxiter = 100
lambda_reg = 0.1 # To avoid overfitting
myargs = (input_layer_size, hidden_layer_size, num_labels, X_train, y_train,
lambda_reg)

# Calling minimize function to minimize cost function and to train weights
results = minimize(neural_network, x0=initial_nn_params, args=myargs,
                   options={'disp': True, 'maxiter': maxiter}, method="L-BFGS-B", jac=True)
```

```
nn_params = results["x"] # Trained Theta is extracted

# Weights are split back to Theta1, Theta2
Theta1 = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)], (
                                                    hidden_layer_size, input_layer_size + 1))
# shape = (100, 785)
Theta2 = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],
                                        (num_labels, hidden_layer_size + 1)) # shape = (10,
101)

# Checking test set accuracy of our model
pred = predict(Theta1, Theta2, X_test)
print('Test Set Accuracy: {:f}'.format((np.mean(pred == y_test) * 100)))

# Checking train set accuracy of our model
pred = predict(Theta1, Theta2, X_train)
print('Training Set Accuracy: {:f}'.format((np.mean(pred == y_train) * 100)))

# Evaluating precision of our model
true_positive = 0
for i in range(len(pred)):
        if pred[i] == y_train[i]:
                true_positive += 1
false_positive = len(y_train) - true_positive
print('Precision =', true_positive/(true_positive + false_positive))

# Saving Thetas in .txt file
np.savetxt('Theta1.txt', Theta1, delimiter=' ')
```

```
np.savetxt('Theta2.txt', Theta2, delimiter=' ')
```

RandInitialise.py

It randomly initializes theta between a range of [-epsilon, +epsilon].

```
import numpy as np


def initialise(a, b):
    epsilon = 0.15
    c = np.random.rand(a, b + 1) * (2 * epsilon) - epsilon
    # Randomly initialises values of thetas between [-epsilon, +epsilon]
    return c
```

Model.py

The function performs feed-forward and backpropagation.

Forward propagation: Input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes it to the successive layer. We will use the sigmoid function as our “activation function”.

Backward propagation: It is the practice of fine-tuning the weights of a neural net based on the error rate obtained in the previous iteration.

It also calculates cross-entropy costs for checking the errors between the prediction and original values. In the end, the gradient is calculated for the optimization objective.

```
import numpy as np


def neural_network(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, lamb):
    # Weights are split back to Theta1, Theta2
    Theta1 = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)],
                        (hidden_layer_size, input_layer_size + 1))
    Theta2 = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],
```

```
                                        (num_labels, hidden_layer_size + 1))

        # Forward propagation
        m = X.shape[0]
        one_matrix = np.ones((m, 1))
        X = np.append(one_matrix, X, axis=1) # Adding bias unit to first layer
        a1 = X
        z2 = np.dot(X, Theta1.transpose())
        a2 = 1 / (1 + np.exp(-z2)) # Activation for second layer
        one_matrix = np.ones((m, 1))
        a2 = np.append(one_matrix, a2, axis=1) # Adding bias unit to hidden layer
        z3 = np.dot(a2, Theta2.transpose())
        a3 = 1 / (1 + np.exp(-z3)) # Activation for third layer

        # Changing the y labels into vectors of boolean values.
        # For each label between 0 and 9, there will be a vector of length 10
        # where the ith element will be 1 if the label equals i
        y_vect = np.zeros((m, 10))
        for i in range(m):
                y_vect[i, int(y[i])] = 1

        # Calculating cost function
       J = (1 / m) * (np.sum(np.sum(-y_vect * np.log(a3) - (1 - y_vect) * np.log(1 - a3)))) + (lamb / (2 * m)) * (
                                sum(sum(pow(Theta1[:, 1:], 2))) + sum(sum(pow(Theta2[:, 1:], 2))))

        # backprop
        Delta3 = a3 - y_vect
```

```
    Delta2 = np.dot(Delta3, Theta2) * a2 * (1 - a2)
    Delta2 = Delta2[:, 1:]

    # gradient
    Theta1[:, 0] = 0
    Theta1_grad = (1 / m) * np.dot(Delta2.transpose(), a1) + (lamb / m) * Theta1
    Theta2[:, 0] = 0
    Theta2_grad = (1 / m) * np.dot(Delta3.transpose(), a2) + (lamb / m) * Theta2
    grad = np.concatenate((Theta1_grad.flatten(), Theta2_grad.flatten()))

    return J, grad
```

Prediction.py

It performs forward propagation to predict the digit.

```
import numpy as np


def predict(Theta1, Theta2, X):
    m = X.shape[0]
    one_matrix = np.ones((m, 1))
    X = np.append(one_matrix, X, axis=1) # Adding bias unit to first layer
    z2 = np.dot(X, Theta1.transpose())
    a2 = 1 / (1 + np.exp(-z2)) # Activation for second layer
    one_matrix = np.ones((m, 1))
    a2 = np.append(one_matrix, a2, axis=1) # Adding bias unit to hidden layer
    z3 = np.dot(a2, Theta2.transpose())
    a3 = 1 / (1 + np.exp(-z3)) # Activation for third layer
```

```
    p = (np.argmax(a3, axis=1)) # Predicting the class on the basis of max value of hypothesis
    return p
```

GUI.py

It launches a GUI for writing digits. The image of the digit is stored in the same directory after converting it to grayscale and reducing the size to (28 X 28) pixels.

```
from tkinter import *
import numpy as np
from PIL import ImageGrab
from Prediction import predict

window = Tk()
window.title("Handwritten digit recognition")
l1 = Label()


def MyProject():
    global l1

    widget = cv
    # Setting co-ordinates of canvas
    x = window.winfo_rootx() + widget.winfo_x()
    y = window.winfo_rooty() + widget.winfo_y()
    x1 = x + widget.winfo_width()
    y1 = y + widget.winfo_height()

    # Image is captured from canvas and is resized to (28 X 28) px
    img = ImageGrab.grab().crop((x, y, x1, y1)).resize((28, 28))
```

```
        # Converting rgb to grayscale image
        img = img.convert('L')

        # Extracting pixel matrix of image and converting it to a vector of (1, 784)
        x = np.asarray(img)
        vec = np.zeros((1, 784))
        k = 0
        for i in range(28):
            for j in range(28):
                vec[0][k] = x[i][j]
                k += 1

        # Loading Thetas
        Theta1 = np.loadtxt('Theta1.txt')
        Theta2 = np.loadtxt('Theta2.txt')

        # Calling function for prediction
        pred = predict(Theta1, Theta2, vec / 255)

        # Displaying the result
        l1 = Label(window, text="Digit = " + str(pred[0]), font=('Algerian', 20))
        l1.place(x=230, y=420)


lastx, lasty = None, None
```

```
# Clears the canvas
def clear_widget():
    global cv, l1
    cv.delete("all")
    l1.destroy()


# Activate canvas
def event_activation(event):
    global lastx, lasty
    cv.bind('<B1-Motion>', draw_lines)
    lastx, lasty = event.x, event.y


# To draw on canvas
def draw_lines(event):
    global lastx, lasty
    x, y = event.x, event.y
    cv.create_line((lastx, lasty, x, y), width=30, fill='white', capstyle=ROUND, smooth=TRUE, splinesteps=12)
    lastx, lasty = x, y


# Label
L1 = Label(window, text="Handwritten Digit Recoginition", font=('Algerian', 25), fg="blue")
L1.place(x=35, y=10)

# Button to clear canvas
```

```
b1 = Button(window, text="1. Clear Canvas", font=('Algerian', 15), bg="orange", fg="black", command=clear_widget)

b1.place(x=120, y=370)


# Button to predict digit drawn on canvas

b2 = Button(window, text="2. Prediction", font=('Algerian', 15), bg="white", fg="red", command=MyProject)

b2.place(x=320, y=370)


# Setting properties of canvas

cv = Canvas(window, width=350, height=290, bg='black')

cv.place(x=120, y=70)


cv.bind('<Button-1>', event_activation)

window.geometry("600x500")

window.mainloop()
```

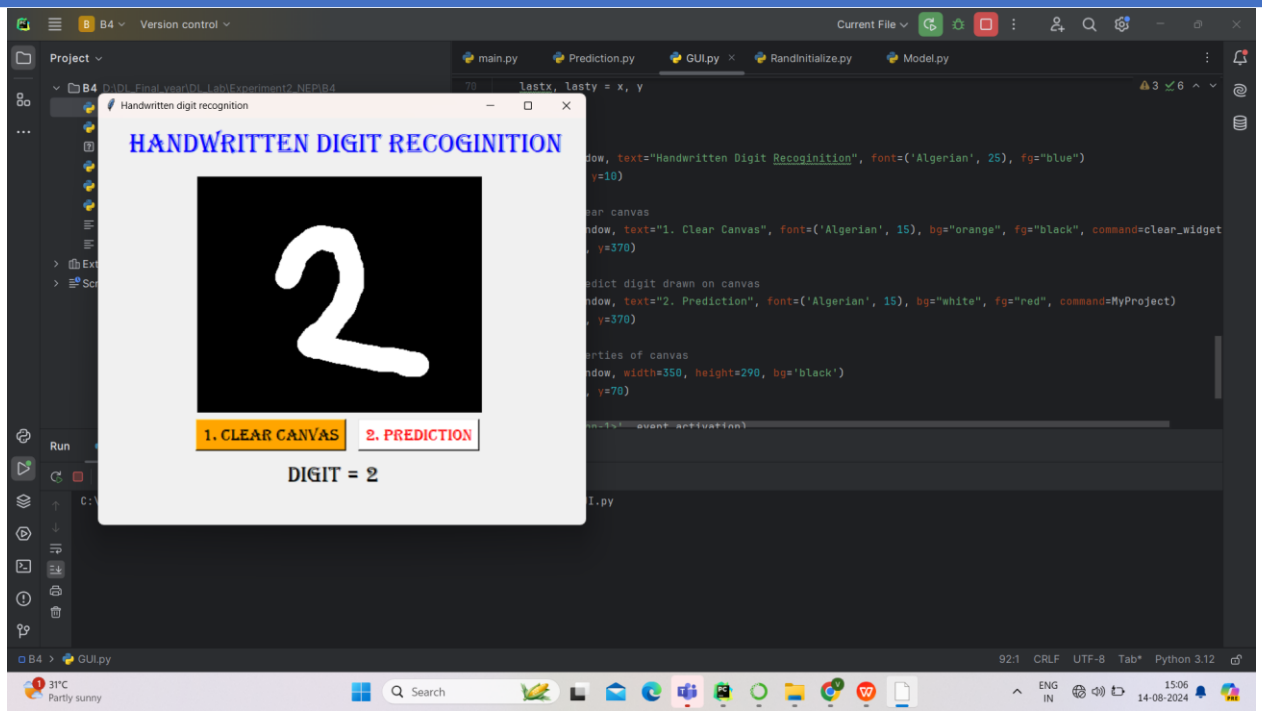
Result:

Training set accuracy of 99.440000%

Test set accuracy of 97.320000%

Precision of 0.9944

Output:



Viva Questions:

- What is Artificial Neural Network?
- What is the size of mnist dataset image?
- What are the main challenges associated with training autoencoders, and how can they be addressed to improve model performance and generalization?

Sr. No.	Question	CO Mapped
1	What is Artificial Neural Network?	1ADPC314_1
2	What is the size of mnist dataset image?	1ADPC314_1
3	How training size and testing size affects model performance?	1ADPC314_3
4	How to import module in python?	1ADPC314_4
5	What are the main challenges associated with training autoencoders, and how can they be addressed to improve model performance and generalization?	1ADPC314_4

Conclusion:

The Artificial Neural Network recognizes handwritten digit it is displayed using GUI.