

**University Of Petroleum & Energy Studies,**

**Dehradun**

**Academic Year 2023-24**

**Department: School of Computer Science**

**NEURAL NETWORK**

**Submitted to- Submitted by-**

**Dr. NIHARIKA SINGH Nihar**

**Batch -1(H)(AI&ML)**

**Sap id-500091867**

**ROLL NO:-R21422105254**

|  |  |  |
| --- | --- | --- |
| **S.**  **No.** | **Experiment** | **Date** |
| 1 | Introduction to PyTorch | 16/08/23 |
| 2 | Performing Activation Function by using Sigmoid and Tanh functions. | 06/09/23 |
| 3 | To Create First Neural Network | 20/09/23 |
| 4 | Implementation of RNN(Recurrent Neural Network). | 04/10/23 |
| 5 | Implementation of Adaline(Adaptive Linear Neuron) | 11/10/23 |
| 6 | Implementation of CNN(Convolution Neural Network). | 18/10/23 |
| 7 | Implementation of CNN from Scatch | 01/11/23 |
| 8 | Implementation of RNN from Scatch | 08/11/23 |
| 9 | Creating NN From Scratch(Project) |  |
|  |  |  |
|  |  |  |
|  |  |  |

Experiment-1

Aim:- Introduction to PyTorch

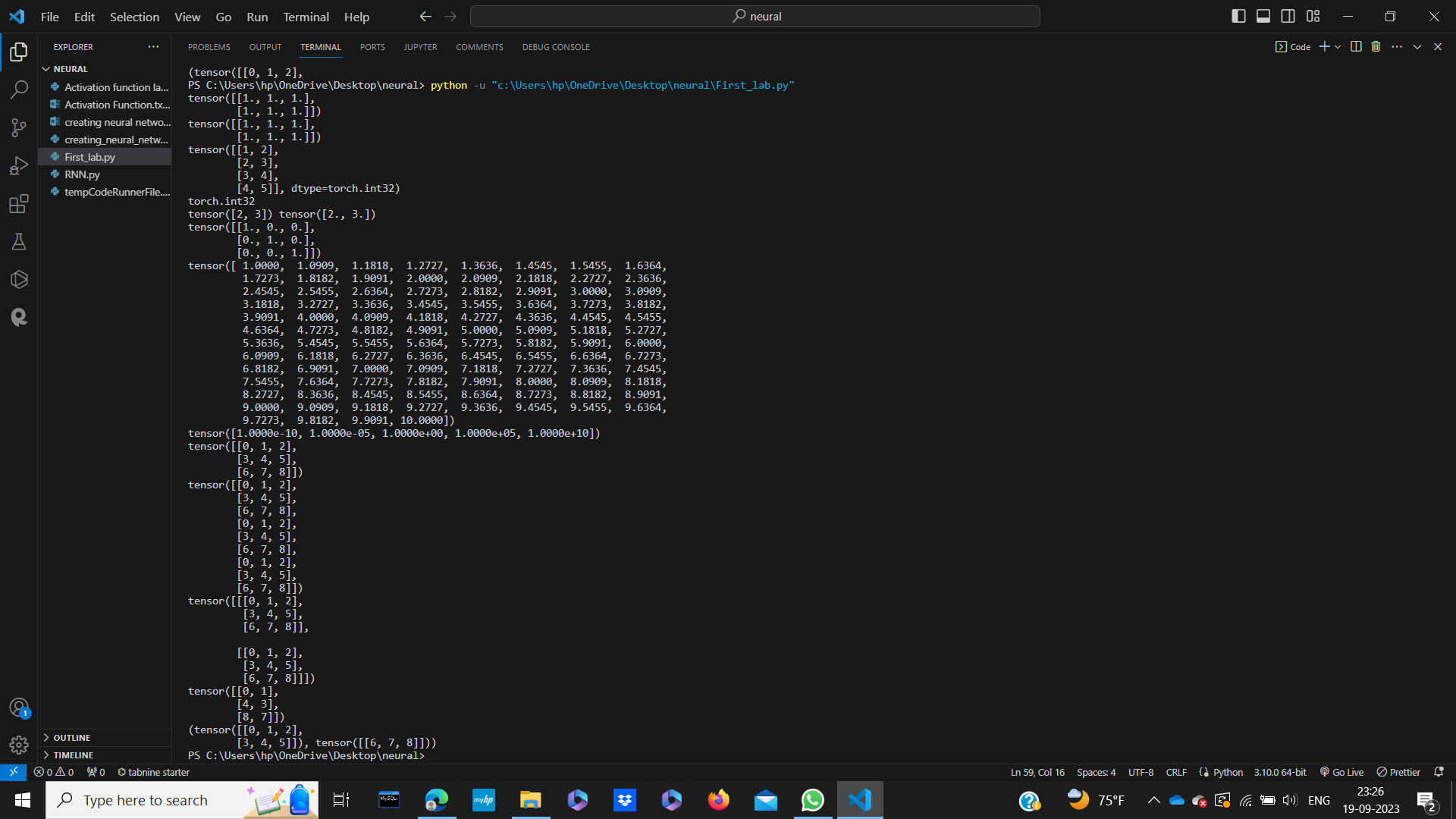
PyToch:- PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is a library of python language used for building deep learning project. It provide core data structure and tensor which work similar to numpy array. PyTorch provides a flexible and dynamic computational graph, which makes it popular among researchers and developers for building and training neural networks.

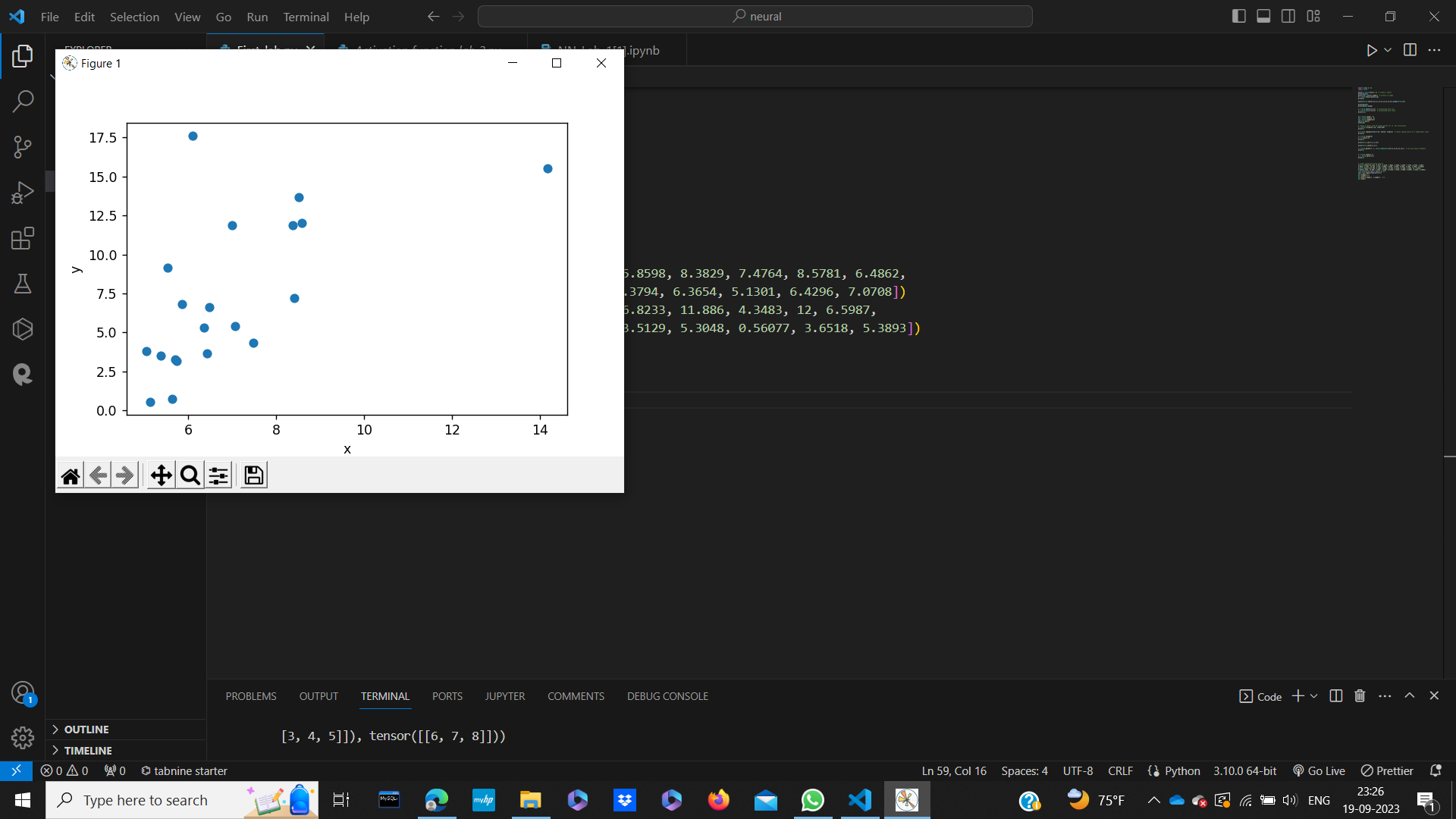
Tensor:- It is a array or data structure which is use to store or collect number which is accessible individual using index. PyToch tensor can convert in numpy and vice versa.

**Code:-**

|  |
| --- |
| **import numpy as np**  **import torch**  **points = torch.ones(2, 3) # create a tensor**  **print(points)**  **points\_np = points.numpy() # convert to numpy**  **b = torch.tensor(points\_np)**  **print(b)**  **points=torch.tensor([[1,2],[2,3],[3,4],[4,5]],dtype=torch.int)**  **print(points)**  **print(points.dtype)**  **v = torch.tensor([2,3]) # initialised with list**  **s = torch.Tensor([2,3]) # initialised with float**  **print(v,s)**  **v1 = torch.rand(2, 3)**  **v2 = torch.randn(2, 3)**  **v3 = torch.randperm(4)**  **id=torch.eye(3)**  **print(id)**  **# Create a Tensor with 10 linear points for (1, 10) inclusively**  **v = torch.linspace(1,10, steps=100)**  **print(v)**  **a = torch.logspace(start=-10, end=10, steps=5) # values spaced evenly on a logarithmic scale**  **print(a)**  **v = torch.arange(9)**  **v = v.view(3,3)**  **print(v)**  **print(torch.cat((v,v,v),0))**  **print(torch.stack((v,v)))**  **r = torch.gather(v, 1, torch.LongTensor([[0,1],[1,0],[2,1]])) # row wise select elements**  **print(r)**  **r = torch.chunk(v,2)**  **s = torch.split(v,2)**  **print(r)**  **# linear regression using pytorch**  **x=torch.tensor([6.1101, 5.5277, 8.5186, 7.0032, 5.8598, 8.3829, 7.4764, 8.5781, 6.4862,**  **5.0546, 5.7107, 14.164, 5.734, 8.4084, 5.6407, 5.3794, 6.3654, 5.1301, 6.4296, 7.0708])**  **y=torch.tensor([17.592, 9.1302, 13.662, 11.854, 6.8233, 11.886, 4.3483, 12, 6.5987,**  **3.8166,3.2522, 15.505, 3.1551, 7.2258, 0.71618, 3.5129, 5.3048, 0.56077, 3.6518, 5.3893])**  **from matplotlib import pyplot as plt**  **fig = plt.figure(figsize=(6,4))**  **plt.xlabel("x")**  **plt.ylabel("y")**  **plt.plot(x.numpy(), y.numpy(), 'o')**  **plt.show()** |

**Output:-**

****

****

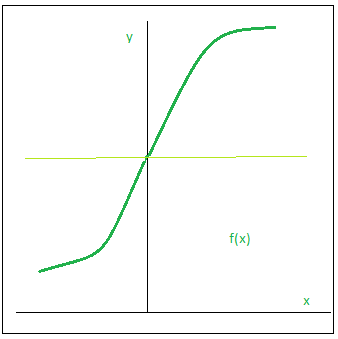
Experiment – 2

Aim:- Performing Activation Function by using Sigmoid and Tanh functions.

Activation Function:- Activation Function is a mathematical function which is applied at the output node to introduce non-linearity in a Artificial Neuron. It is used to determine weather neuron is active or not based on input values.

Activation Function used in this program are:-

Sigmoid:- It is most used activation function in Neural Network and Logistic Regression. It’s Output range between 0 and 1. It’s look like a S-curve because its value increases from 0 to 1

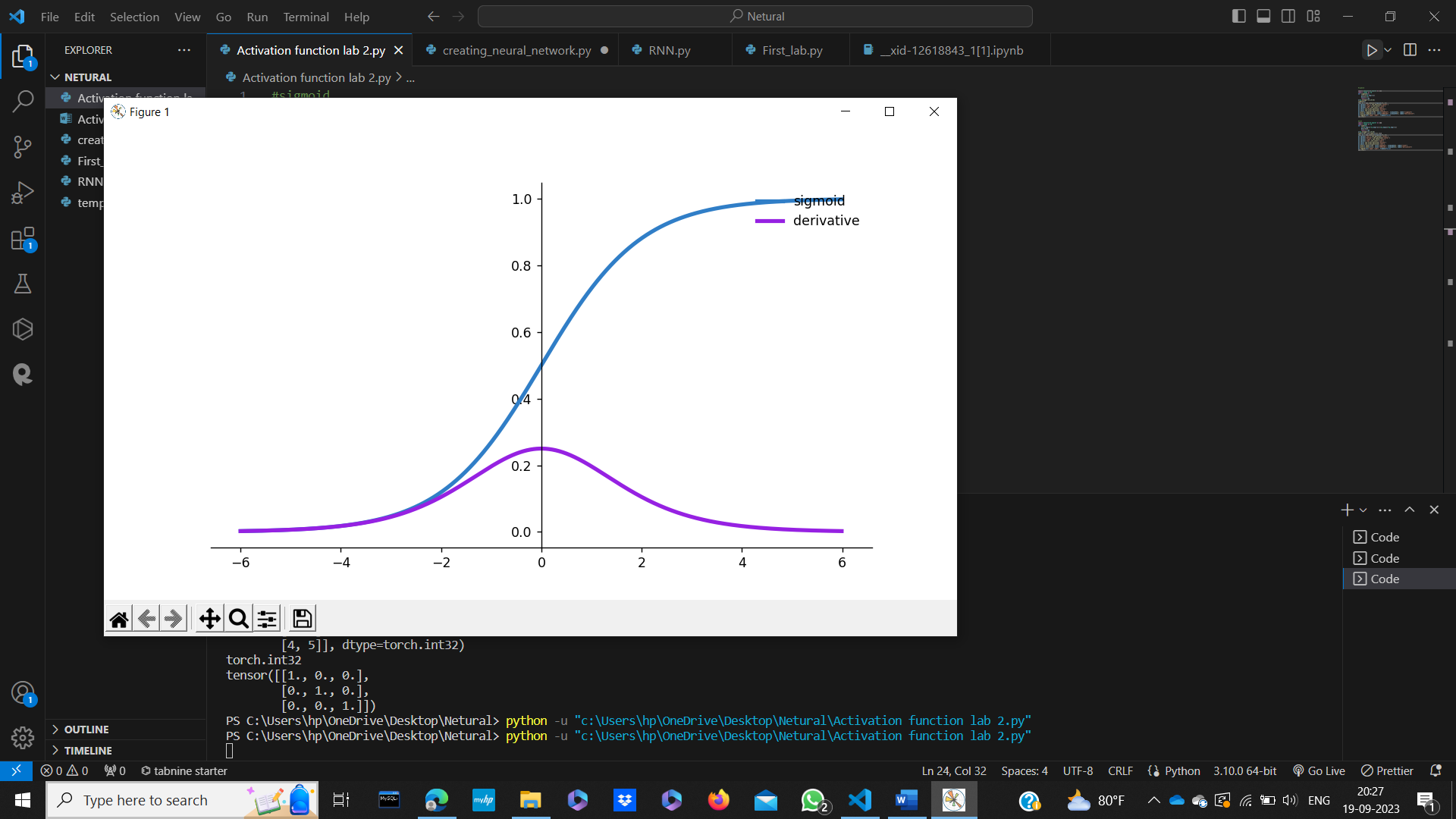


Source:- Geeks For Geeks.

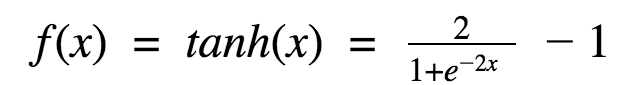
**Code:-**

|  |
| --- |
| #sigmoid  import matplotlib.pyplot as plt  import numpy as np  def sigmoid(x):  s=1/(1+np.exp(-x))  ds=s\*(1-s)  return s,ds  x=np.arange(-6,6,0.01)  sigmoid(x)  fig, ax = plt.subplots(figsize=(9, 5))  ax.spines['left'].set\_position('center')  ax.spines['right'].set\_color('none')  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  ax.yaxis.set\_ticks\_position('left')  ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")  ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")  ax.legend(loc="upper right", frameon=False)  plt.show() |

**Output:-**

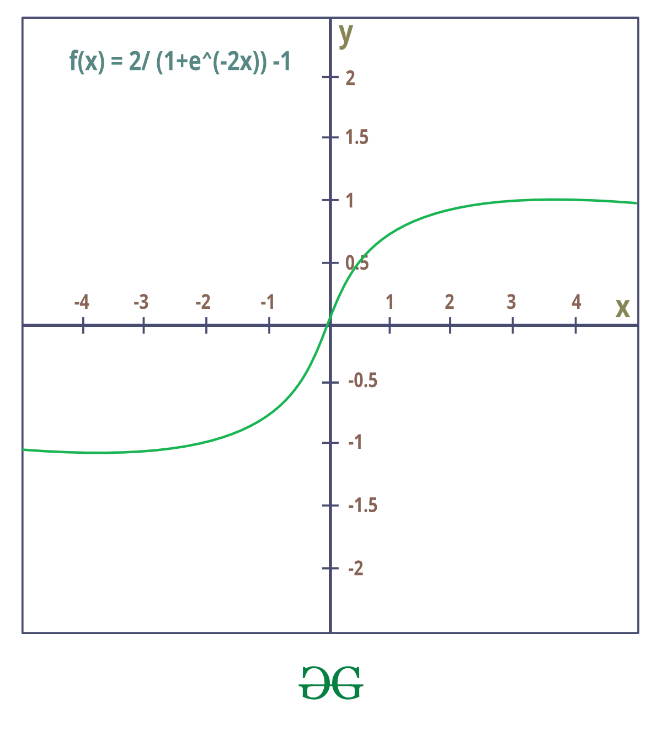


Tanh:- It is another activation function in neural network, whose value lies between -1 and 1. It is also S-shape curve.



Or

tanh(x) = (e^(x) - e^(-x)) / (e^(x) + e^(-x))

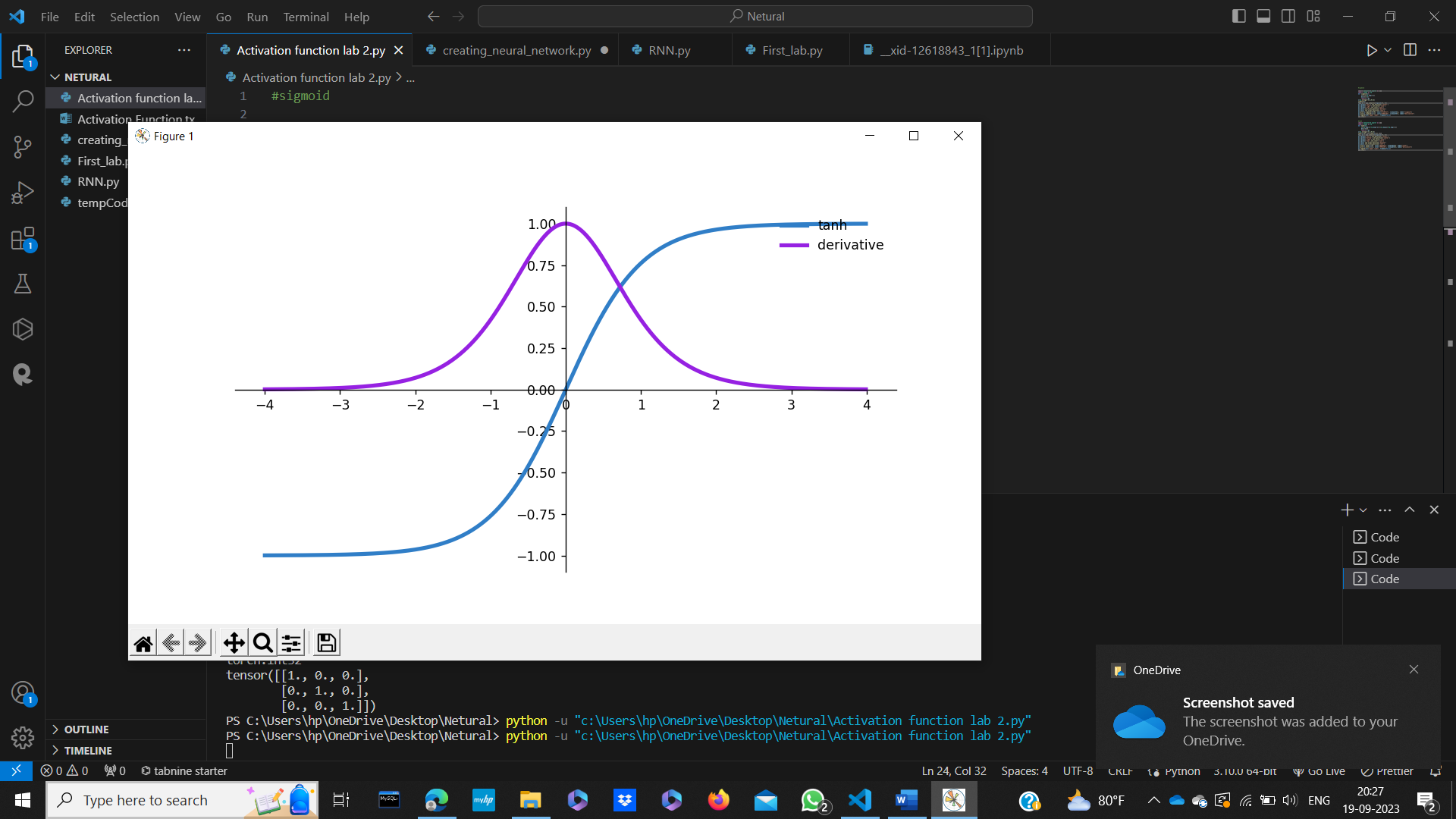


Source:-Geeks For Geeks

**Code:-**

|  |
| --- |
| #tanh  import matplotlib.pyplot as plt  import numpy as np  def tanh(x):  t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))  dt=1-t\*\*2  return t,dt  z=np.arange(-4,4,0.01)  tanh(z)[0].size,tanh(z)[1].size  fig, ax = plt.subplots(figsize=(9, 5))  ax.spines['left'].set\_position('center')  ax.spines['bottom'].set\_position('center')  ax.spines['right'].set\_color('none')  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  ax.yaxis.set\_ticks\_position('left')  ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")  ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")  ax.legend(loc="upper right", frameon=False)  plt.show() |

**Output:-**



**Learning:-** So, in this experiment we have learn how to implement Sigmoid and Tanh function numpy and math plot library.

Experiment-3

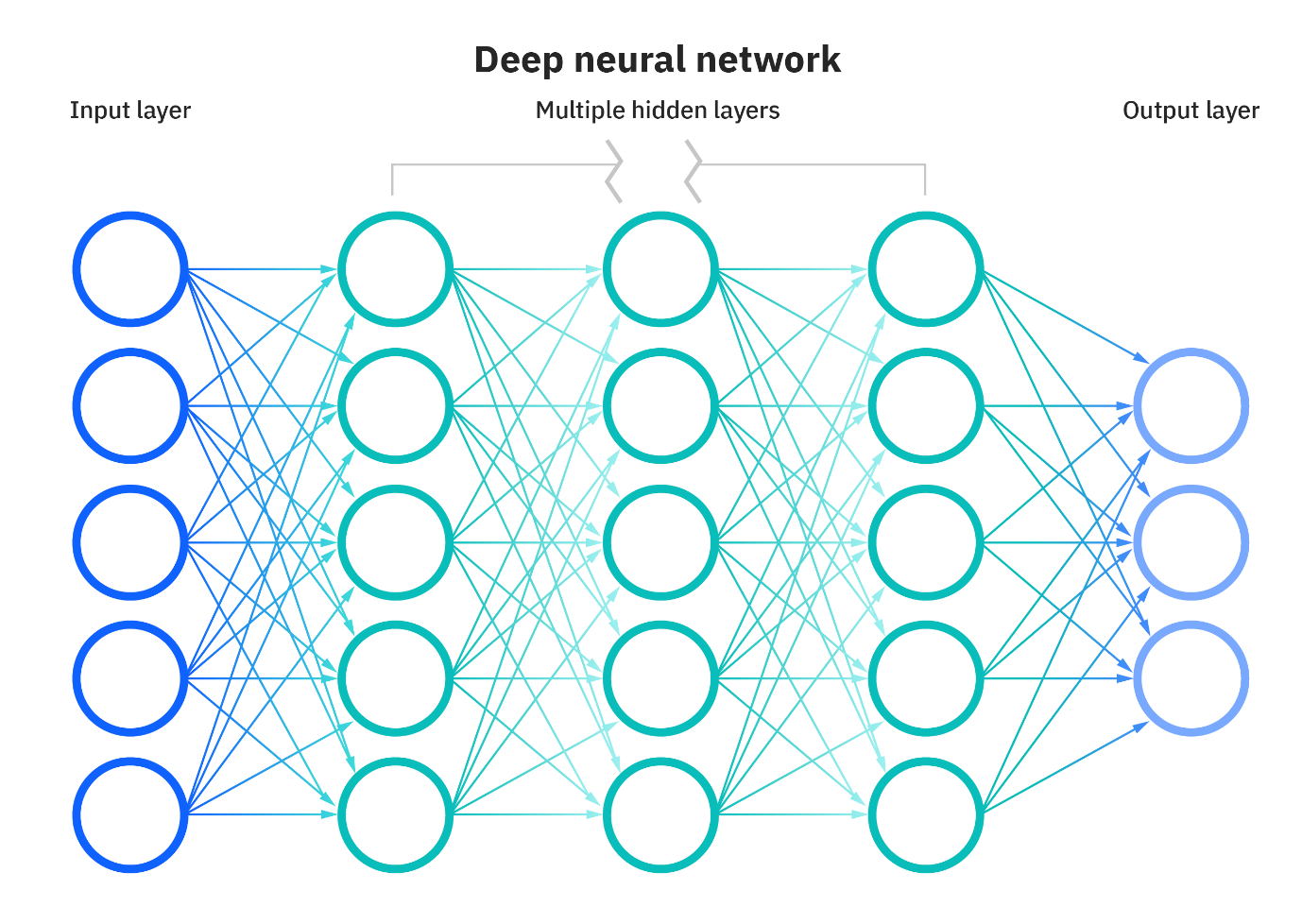
Aim:- To Create First Neural Network

Neural Network:- It is an artificial network or model which is inspired by human brain neural network. It take one or more input to provide desire output.

There are two types of Neural Network:-

|  |  |
| --- | --- |
| Biological Neural Network | Artificial Neural Network |
| BNN is made up of dendrites, synapse ,axon and cell body. | ANN is made up of input, weight, output and hidden layer. |
| It is simple but low speed. | It is complex but high speed. |
| It is connected in network. | It is organised in layers. |

**Artificial Neural Network:-**



Source:- IBM

Input Layer:- It receive raw data as an input.

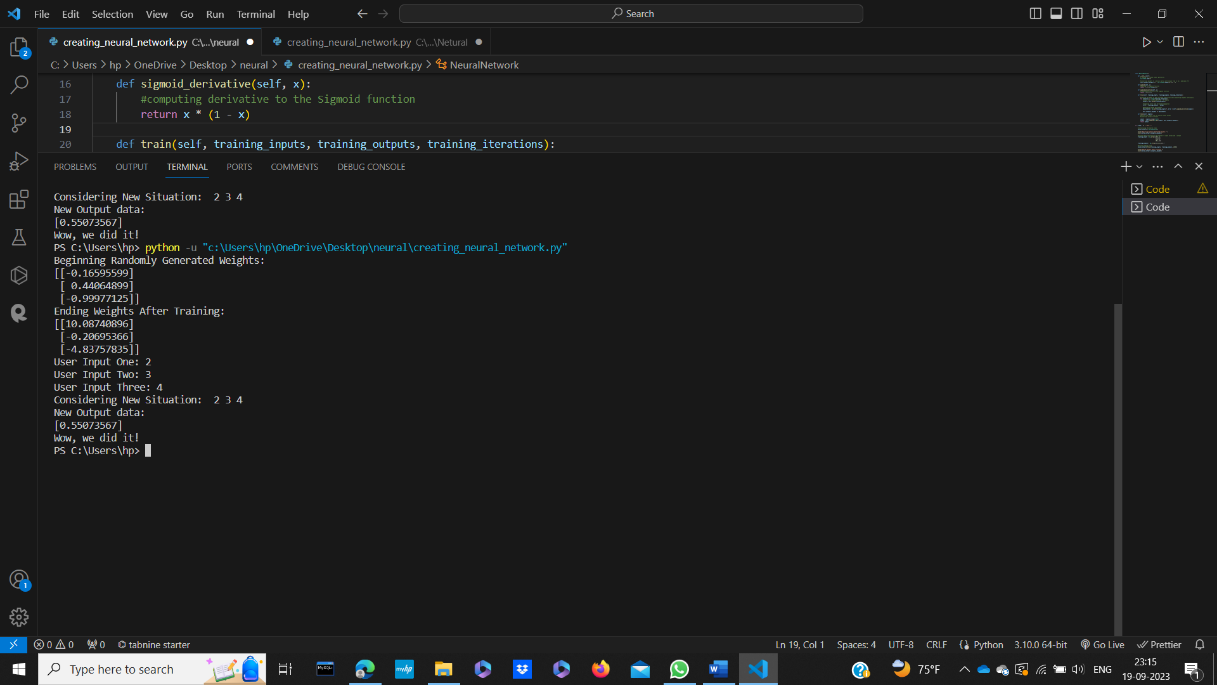
Hidden Layer:- It is intermediary layer between the input and output layer. A simple neural network can contain multiple hidden layer. This layer is use to perform computation on input data by providing weight and activation function.

Output Layer:- This is the final layer of the neural network.

**Code:-**

|  |
| --- |
| import numpy as np  class NeuralNetwork():    def \_\_init\_\_(self):  # seeding for random number generation  np.random.seed(1)    #converting weights to a 3 by 1 matrix with values from -1 to 1 and mean of 0  self.synaptic\_weights = 2 \* np.random.random((3, 1)) - 1  def sigmoid(self, x):  #applying the sigmoid function  return 1 / (1 + np.exp(-x))  def sigmoid\_derivative(self, x):  #computing derivative to the Sigmoid function  return x \* (1 - x)  def train(self, training\_inputs, training\_outputs, training\_iterations):    #training the model to make accurate predictions while adjusting weights continually  for iteration in range(training\_iterations):  #siphon the training data via the neuron  output = self.think(training\_inputs)  #computing error rate for back-propagation  error = training\_outputs - output    #performing weight adjustments  adjustments = np.dot(training\_inputs.T, error \* self.sigmoid\_derivative(output))  self.synaptic\_weights += adjustments  def think(self, inputs):  #passing the inputs via the neuron to get output  #converting values to floats    inputs = inputs.astype(float)  output = self.sigmoid(np.dot(inputs, self.synaptic\_weights))  return output  if \_\_name\_\_ == "\_\_main\_\_":  #initializing the neuron class  neural\_network = NeuralNetwork()  print("Beginning Randomly Generated Weights: ")  print(neural\_network.synaptic\_weights)  #training data consisting of 4 examples--3 input values and 1 output  training\_inputs = np.array([[0,0,1],  [1,1,1],  [1,0,1],  [0,1,1]])  training\_outputs = np.array([[0,1,1,0]]).T  #training taking place  neural\_network.train(training\_inputs, training\_outputs, 15000)  print("Ending Weights After Training: ")  print(neural\_network.synaptic\_weights)  user\_input\_one = str(input("User Input One: "))  user\_input\_two = str(input("User Input Two: "))  user\_input\_three = str(input("User Input Three: "))    print("Considering New Situation: ", user\_input\_one, user\_input\_two, user\_input\_three)  print("New Output data: ")  print(neural\_network.think(np.array([user\_input\_one, user\_input\_two, user\_input\_three])))  print("Wow, we did it!") |

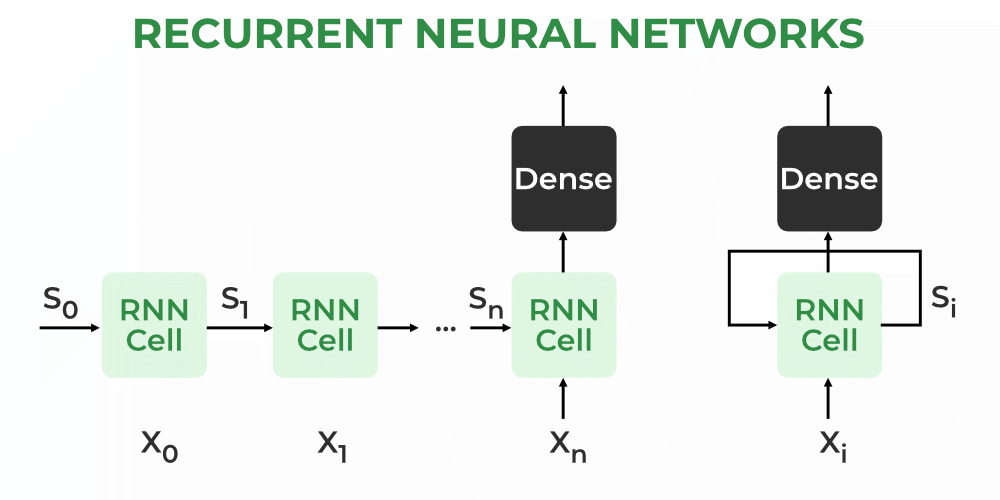
**Output:-**

****

Experiment-4

Aim:- Implementation of RNN(Recurrent Neural Network).

RNN:- It is a type of artificial neural network designed for processing sequences of data. It is use for task that involves time series data. RNN has recurrent connection, which allow them to maintain a hidden state or memory of previous time steps and use it in current input.



Source:- Geeks For Geeks

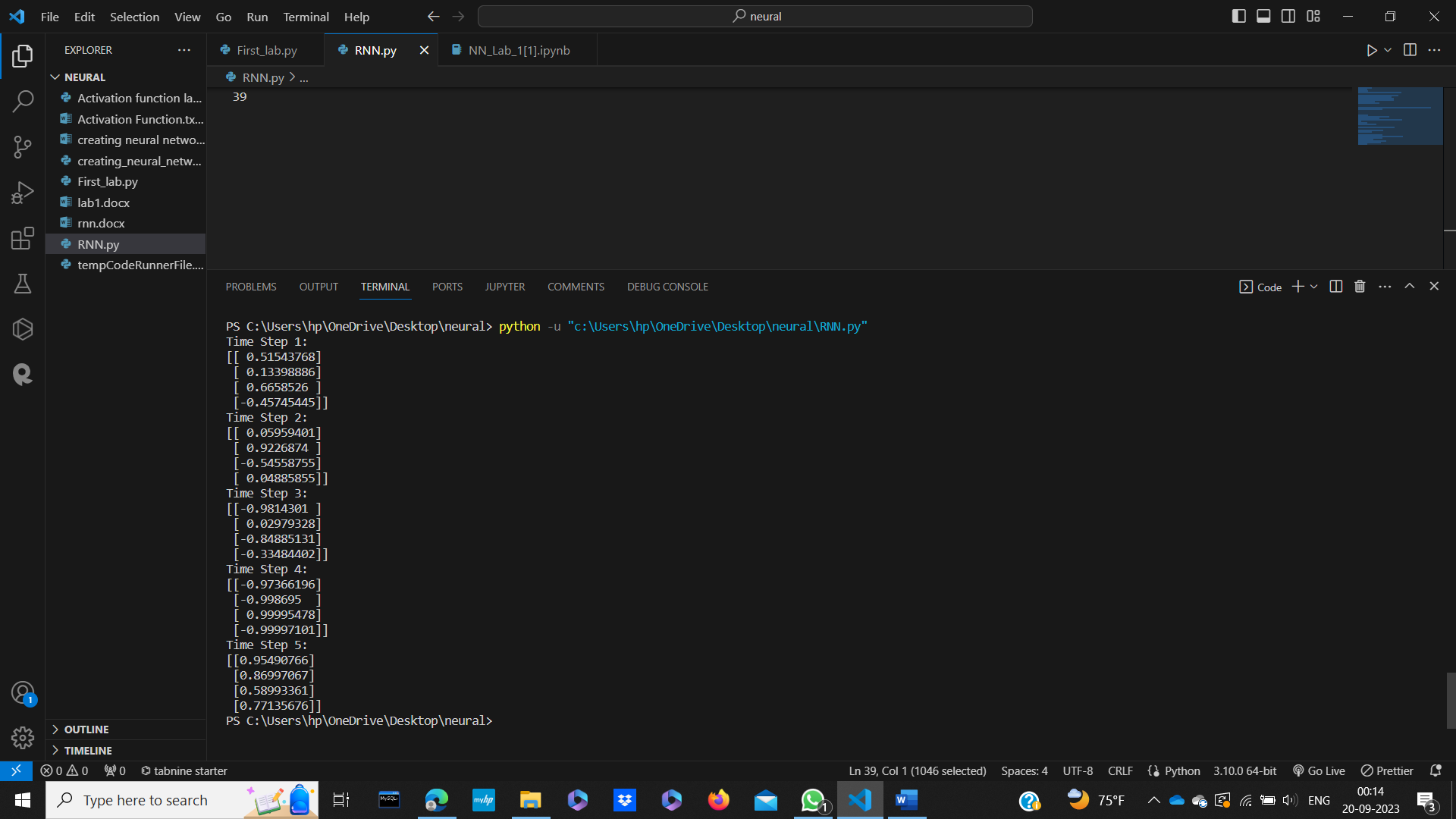
RNN consist of multiple fixed activation function unit, one is for each time step. In this hidden layer signifies the past knowledge that network currently hold at given time step. We mostly use tanh function here.



**Code:-**

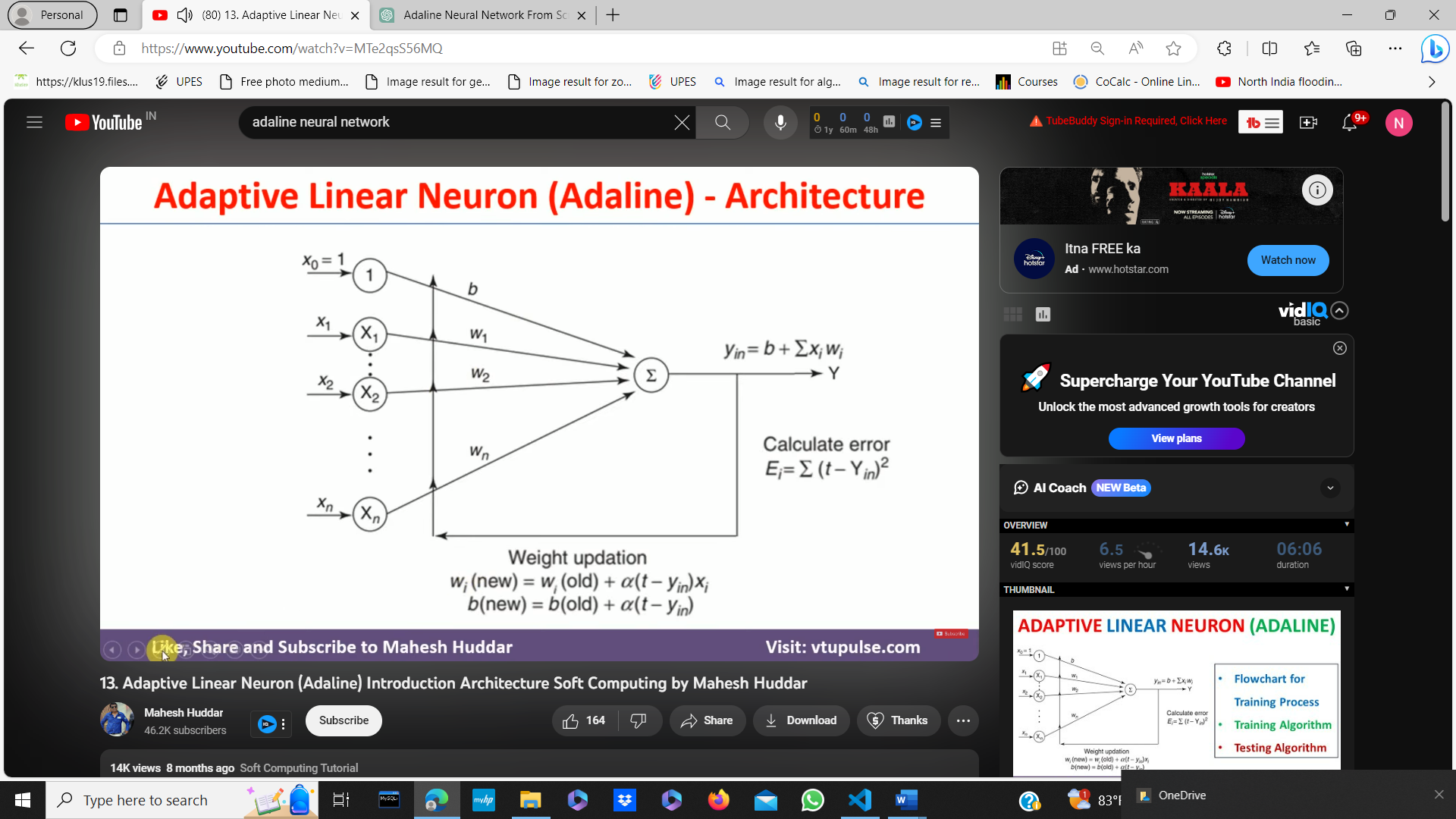
|  |
| --- |
| import numpy as np  input\_size=3  hidden\_size=4  sequence\_length=5 #tell how many time we have to set back  #random select weight for input layer to hidden layer  wxh=np.random.randn(hidden\_size, input\_size)  #random select weight for hiden layer to hidden  whh = np.random.randn(hidden\_size, hidden\_size)  #bias for hidden layer  bh=np.zeros((hidden\_size,1))  #next hidden state or we can say previous hidden state because all previous data is going to be  h\_prev=np.zeros((hidden\_size,1))  #forward pass  def rnn\_forward(x, h\_prev, Wxh, Whh, bh):  # Calculate hidden state  h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h\_prev) + bh)    return h  #Now generate the input  x = np.random.randn(input\_size, sequence\_length)  #forward pass for each time stamp  hidden\_states = []  for t in range(sequence\_length):  h\_prev = rnn\_forward(x[:, t:t+1], h\_prev, wxh, whh, bh)  hidden\_states.append(h\_prev)  #print hidden states  for t, h in enumerate(hidden\_states):  print(f"Time Step {t+1}:")  print(h) |

**Output:-**



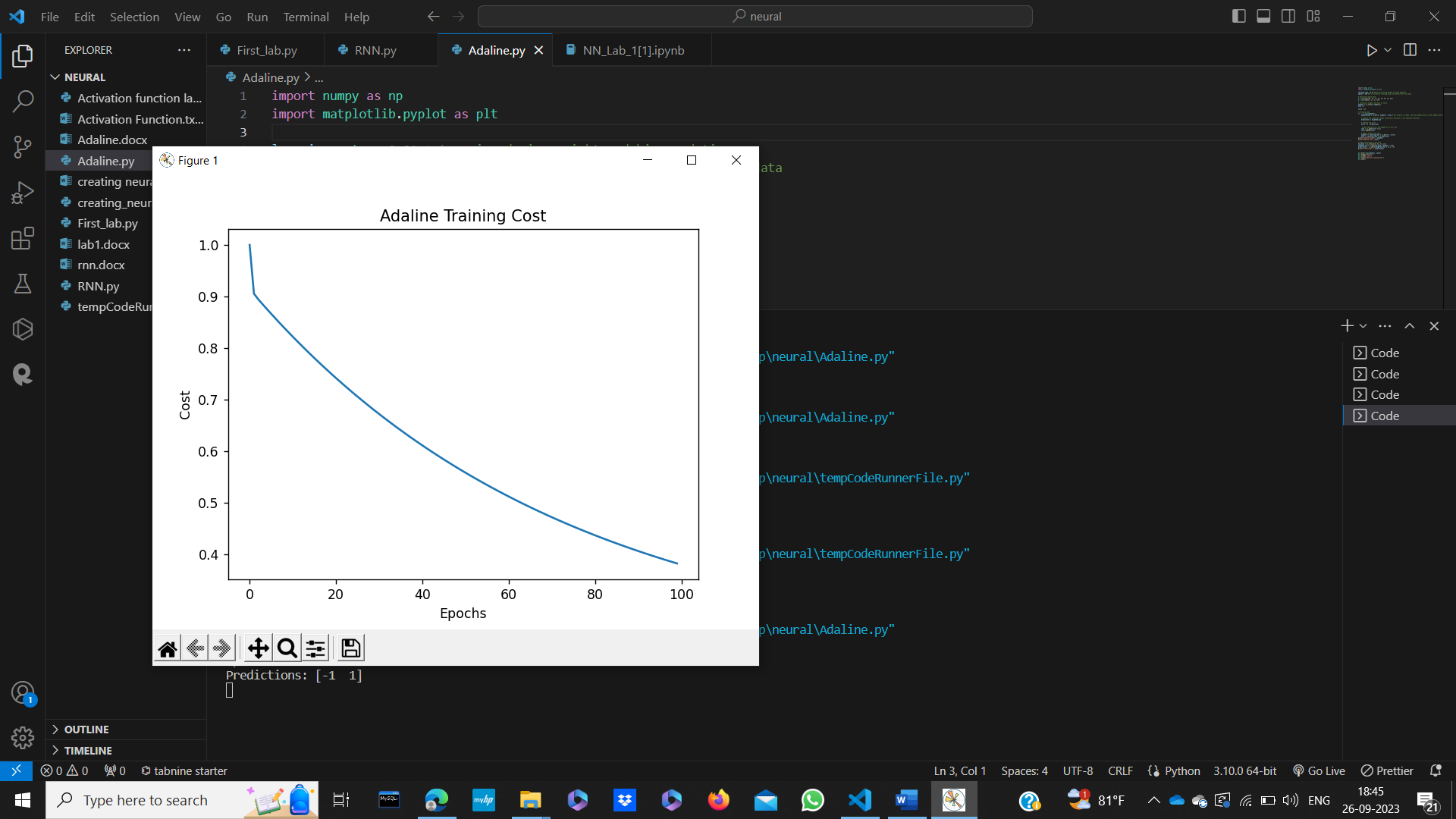
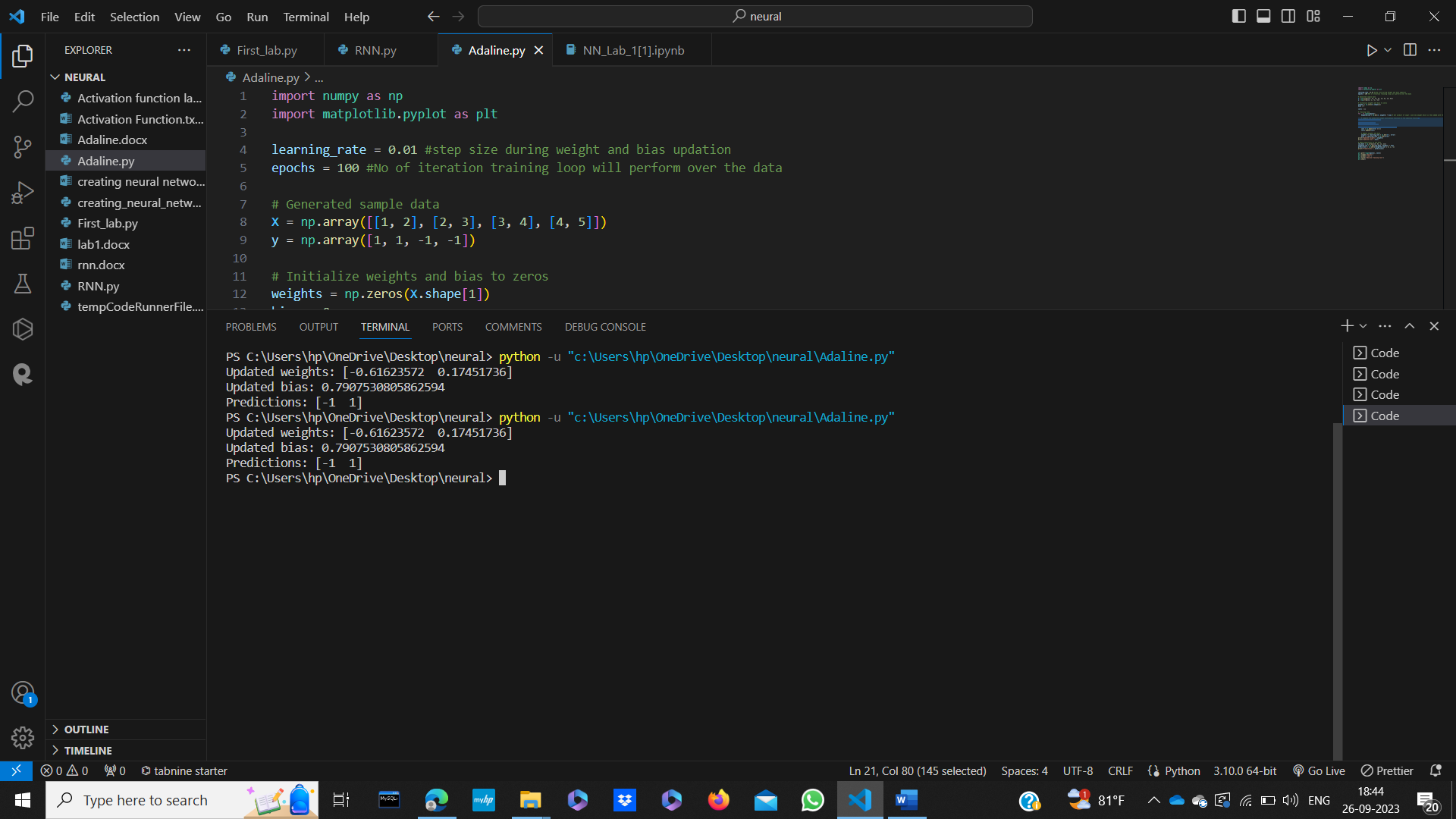
Experiment-5

Aim:- Implementation of Adaline(Adaptive Linear Neuron)

Adaline:- Adaline is a network with a single linear adaptive function or we can say single layer artificial neural network. In this adaptive linear neuron, input and output are in linear manner. In this we use Bipolar activation function with our input value. It make our prediction in 1 or -1.

**Code:-**

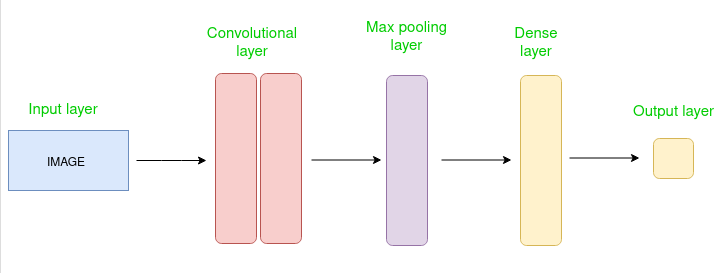
|  |
| --- |
| import numpy as np  import matplotlib.pyplot as plt  learning\_rate = 0.01 #step size during weight and bias updation  epochs = 100 #No of iteration training loop will perform over the data  # Generated sample data  X = np.array([[1, 2], [2, 3], [3, 4], [4, 5]])  y = np.array([1, 1, -1, -1])  # Initialize weights and bias to zeros  weights = np.zeros(X.shape[1])  bias = 0  costs = []  # Training loop  for \_ in range(epochs):  weighted\_sum = np.dot(X, weights) + bias # dot product of input x and new weight which is then added with the bias  # Compute the predicted output (activation function is the identity function)  predictions = weighted\_sum  # Compute the error  error = y - predictions  # mean squared error and append it to the list  cost = np.mean(error \*\* 2)  costs.append(cost)  # Update weights and bias  weights += learning\_rate \* np.dot(X.T, error)  bias += learning\_rate \* np.sum(error)  print("Updated weights:",weights)  print("Updated bias:",bias)  # Make predictions on new data  new\_data = np.array([[5, 6], [1, 1]])  weighted\_sum = np.dot(new\_data, weights) + bias  predictions = np.where(weighted\_sum >= 0, 1, -1)  print("Predictions:", predictions)  plt.plot(range(epochs), costs)  plt.xlabel('Epochs')  plt.ylabel('Cost')  plt.title('Adaline Training Cost')  plt.show() |

**Output:-** 

Experiment-6

Aim:- Implementation of CNN(Convolution Neural Network).

It is a type of deep learning model specifically designed for processing and analyzing visual data, such as images and video. It uses layer of filters to automatically learn and extract features from the input data, enabling tasks like image recognition, object detection etc. CNN has revolutionized computer vision and are widely used in applications like facial reorganization, self-driving cars and medical image analysis.

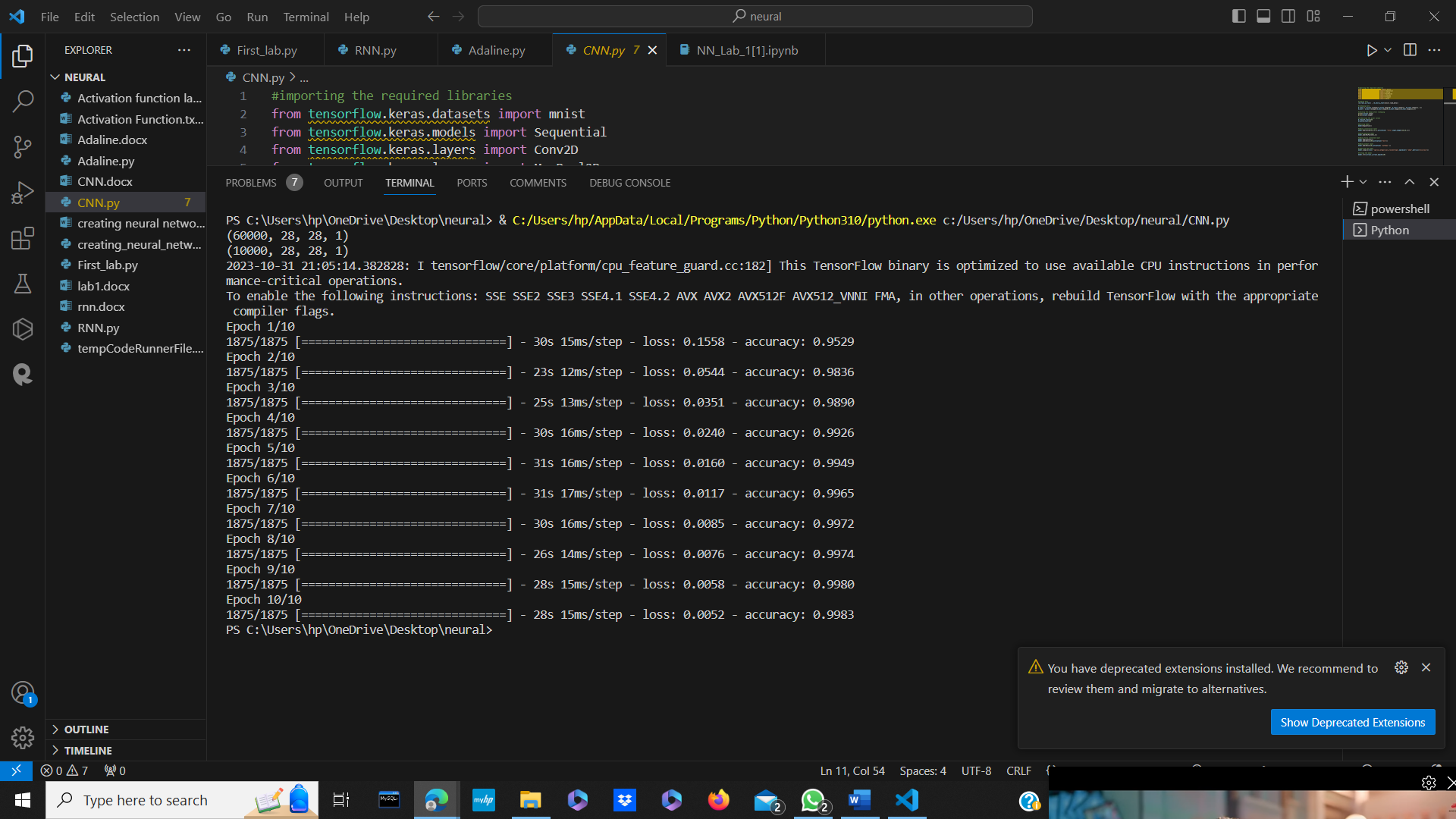


Credit:- Geeksforgeeks.

Code:-

|  |
| --- |
| #importing the required libraries  from tensorflow.keras.datasets import mnist  from tensorflow.keras.models import Sequential  from tensorflow.keras.layers import Conv2D  from tensorflow.keras.layers import MaxPool2D  from tensorflow.keras.layers import Flatten  from tensorflow.keras.layers import Dropout  from tensorflow.keras.layers import Dense  #loading data  (X\_train,y\_train) , (X\_test,y\_test)=mnist.load\_data()  #reshaping data  X\_train = X\_train.reshape((X\_train.shape[0], X\_train.shape[1], X\_train.shape[2], 1))  X\_test = X\_test.reshape((X\_test.shape[0],X\_test.shape[1],X\_test.shape[2],1))  #checking the shape after reshaping  print(X\_train.shape)  print(X\_test.shape)  #normalizing the pixel values  X\_train=X\_train/255  X\_test=X\_test/255  #defining model  model=Sequential()  #adding convolution layer  model.add(Conv2D(32,(3,3),activation= "relu",input\_shape=(28,28,1)))  #adding pooling layer  model.add(MaxPool2D(2,2))  #adding fully connected layer  model.add(Flatten())  model.add(Dense(100,activation="relu"))  #adding output layer  model.add(Dense(10,activation= "softmax" ))  #compiling the model  model.compile(loss= "sparse\_categorical\_crossentropy",optimizer= "adam",metrics=["accuracy"])  #fitting the model  model.fit(X\_train,y\_train,epochs=10) |

Output:-



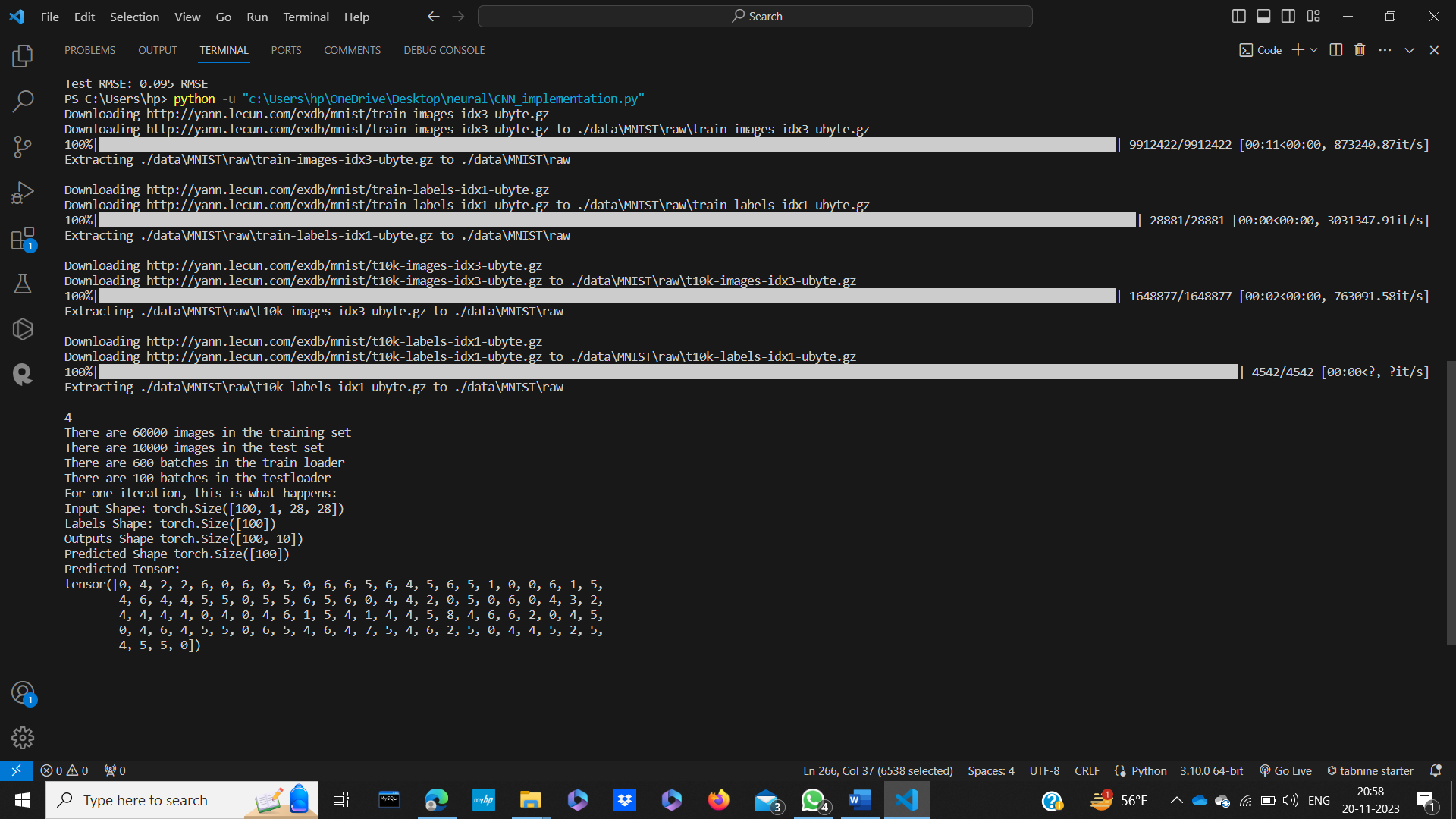
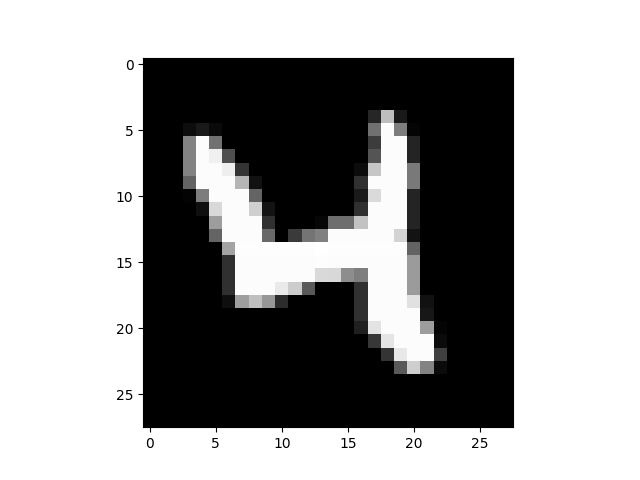
Experiment-7

Aim:- Implementation of CNN from Scatch

Code:-

|  |
| --- |
| import torch  import torch.nn as nn  import torchvision.transforms as transforms  import torchvision.datasets as datasets  from torch.autograd import Variable  # Specify the Mean and standard deviation of all the pixels in the MNIST dataset. They are precomputed  mean\_gray = 0.1307  stddev\_gray = 0.3081  #Transform the images to tensors  #Normalize a tensor image with mean and standard deviation. Given mean: (M1,...,Mn) and std: (S1,..,Sn)  #for n channels, this transform will normalize each channel of the input torch.Tensor  #i.e. input[channel] = (input[channel] - mean[channel]) / std[channel]  transforms = transforms.Compose([transforms.ToTensor(),  transforms.Normalize((mean\_gray,), (stddev\_gray,))])  #Load our dataset  train\_dataset = datasets.MNIST(root = './data',  train = True,  transform = transforms,  download = True)  test\_dataset = datasets.MNIST(root = './data',  train = False,  transform = transforms)  import matplotlib.pyplot as plt  random\_image = train\_dataset[20][0].numpy() \* stddev\_gray + mean\_gray  plt.imshow(random\_image.reshape(28, 28), cmap='gray')  plt.show()  print(train\_dataset[20][1]) #Print the corresponding label for the image  #Make the dataset iterable  batch\_size = 100  train\_load = torch.utils.data.DataLoader(dataset = train\_dataset,  batch\_size = batch\_size,  shuffle = True)  test\_load = torch.utils.data.DataLoader(dataset = test\_dataset,  batch\_size = batch\_size,  shuffle = False)  print('There are {} images in the training set'.format(len(train\_dataset)))  print('There are {} images in the test set'.format(len(test\_dataset)))  print('There are {} batches in the train loader'.format(len(train\_load)))  print('There are {} batches in the testloader'.format(len(test\_load)))  #cnn mnist  class CNN(nn.Module):  def \_\_init\_\_(self):  super(CNN,self).\_\_init\_\_()  self.cnn1 = nn.Conv2d(in\_channels = 1, out\_channels = 8, kernel\_size = 3, stride=1, padding = 1)  self.batchnorm1 = nn.BatchNorm2d(8)  self.relu = nn.ReLU()  self.maxpool = nn.MaxPool2d(kernel\_size = 2)  self.cnn2 = nn.Conv2d(in\_channels = 8, out\_channels = 32, kernel\_size = 5, stride = 1, padding = 2)  self.batchnorm2 = nn.BatchNorm2d(32)  self.fc1 = nn.Linear(1568,600)  self.dropout = nn.Dropout(0.5)  self.fc2 = nn.Linear(600,10)  def forward(self, x):  out = self.cnn1(x)  out = self.batchnorm1(out)  out = self.relu(out)  out = self.maxpool(out)  out = self.cnn2(out)  out = self.batchnorm2(out)  out = self.relu(out)  out = self.maxpool(out)  out = out.view(-1,1568)  out = self.fc1(out)  out = self.relu(out)  out = self.dropout(out)  out = self.fc2(out)  return out    model = CNN()  CUDA = torch.cuda.is\_available()  if CUDA:  model = model.cuda()  loss\_fn = nn.CrossEntropyLoss()  optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)  iteration = 0  correct = 0  for i,(inputs,labels) in enumerate (train\_load):  if CUDA:  inputs = inputs.cuda()  labels = labels.cuda()  print("For one iteration, this is what happens:")  print("Input Shape:",inputs.shape)  print("Labels Shape:",labels.shape)  output = model(inputs)  print("Outputs Shape",output.shape)  \_, predicted = torch.max(output, 1)  print("Predicted Shape",predicted.shape)  print("Predicted Tensor:")  print(predicted)  correct += (predicted == labels).sum()  break  #Training the CNN  num\_epochs = 25  train\_loss = []  test\_loss = []  train\_accuracy = []  test\_accuracy = []  for epoch in range(num\_epochs):  correct = 0  iterations = 0  iter\_loss = 0.0  model.train()  for i, (inputs, labels) in enumerate(train\_load):  if CUDA:  inputs = inputs.cuda()  labels = labels.cuda()  outputs = model(inputs)  loss = loss\_fn(outputs, labels)  iter\_loss += loss.item() # Accumulate the loss  optimizer.zero\_grad() # Clear off the gradient in (w = w - gradient)  loss.backward() # Backpropagation  optimizer.step() # Update the weights  \_, predicted = torch.max(outputs, 1)  correct += (predicted == labels).sum()  iterations += 1  # Record the training loss  train\_loss.append(iter\_loss/iterations)  # Record the training accuracy  train\_accuracy.append((100 \* correct / len(train\_dataset)))  #Testing  testing\_loss = 0.0  correct = 0  iterations = 0  model.eval() # Put the network into evaluation mode  for i, (inputs, labels) in enumerate(test\_load):  if CUDA:  inputs = inputs.cuda()  labels = labels.cuda()  outputs = model(inputs)  loss = loss\_fn(outputs, labels) # Calculate the loss  testing\_loss += loss.item()  \_, predicted = torch.max(outputs, 1)  correct += (predicted == labels).sum()  iterations += 1  # Record the Testing loss  test\_loss.append(testing\_loss/iterations)  # Record the Testing accuracy  test\_accuracy.append((100 \* correct / len(test\_dataset)))  print ('Epoch {}/{}, Training Loss: {:.3f}, Training Accuracy: {:.3f}, Testing Loss: {:.3f}, Testing Acc: {:.3f}'  .format(epoch+1, num\_epochs, train\_loss[-1], train\_accuracy[-1],  test\_loss[-1], test\_accuracy[-1]))    # Loss  f = plt.figure(figsize=(10, 10))  plt.plot(train\_loss, label='Training Loss')  plt.plot(test\_loss, label='Testing Loss')  plt.legend()  plt.show()  # Accuracy  f = plt.figure(figsize=(10, 10))  plt.plot(train\_accuracy, label='Training Accuracy')  plt.plot(test\_accuracy, label='Testing Accuracy')  plt.legend()  plt.show()  img = test\_dataset[30][0].resize\_((1, 1, 28, 28))  label = test\_dataset[30][1]  plt.imshow(img.reshape(28, 28), cmap='gray')  plt.show()  model.eval()  if CUDA:  model = model.cuda()  img = img.cuda()  output = model(img)  \_, predicted = torch.max(output,1)  print("Prediction is: {}".format(predicted.item()))  print("Actual is: {}".format(label)) |

Output:-

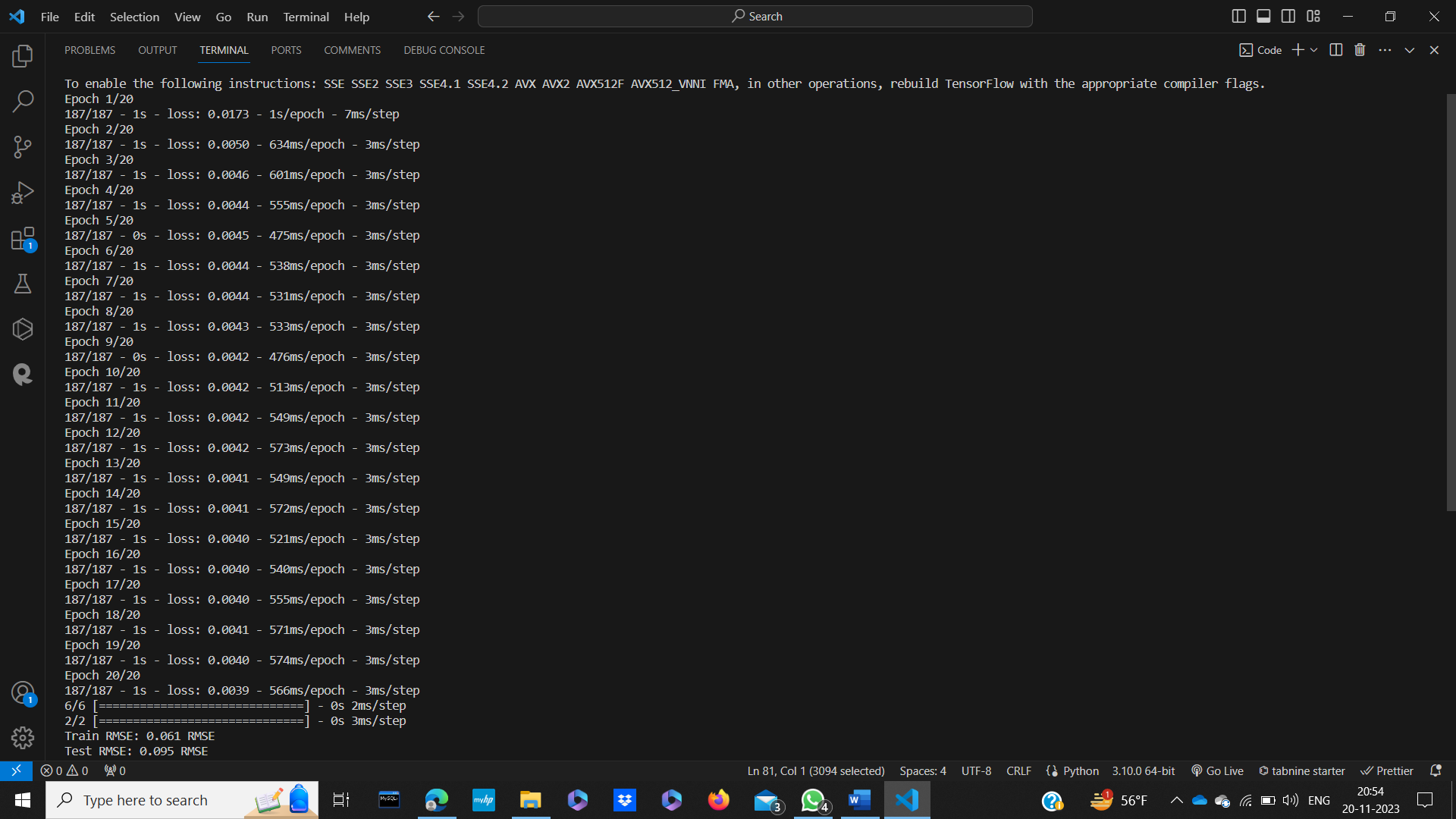
Experiment-8

Aim:- Implementation of RNN from Scatch

Code:-

|  |
| --- |
| from pandas import read\_csv  import numpy as np  from keras.models import Sequential  from keras.layers import Dense, SimpleRNN  from sklearn.preprocessing import MinMaxScaler  from sklearn.metrics import mean\_squared\_error  import math  import matplotlib.pyplot as plt  # Parameter split\_percent defines the ratio of training examples  def get\_train\_test(url, split\_percent=0.8):  df = read\_csv(url, usecols=[1], engine='python')  data = np.array(df.values.astype('float32'))  scaler = MinMaxScaler(feature\_range=(0, 1))  data = scaler.fit\_transform(data).flatten()  n = len(data)  # Point for splitting data into train and test  split = int(n\*split\_percent)  train\_data = data[range(split)]  test\_data = data[split:]  return train\_data, test\_data, data    # Prepare the input X and target Y  def get\_XY(dat, time\_steps):  Y\_ind = np.arange(time\_steps, len(dat), time\_steps)  Y = dat[Y\_ind]  rows\_x = len(Y)  X = dat[range(time\_steps\*rows\_x)]  X = np.reshape(X, (rows\_x, time\_steps, 1))  return X, Y    def create\_RNN(hidden\_units, dense\_units, input\_shape, activation):  model = Sequential()  model.add(SimpleRNN(hidden\_units, input\_shape=input\_shape, activation=activation[0]))  model.add(Dense(units=dense\_units, activation=activation[1]))  model.compile(loss='mean\_squared\_error', optimizer='adam')  return model    def print\_error(trainY, testY, train\_predict, test\_predict):  # Error of predictions  train\_rmse = math.sqrt(mean\_squared\_error(trainY, train\_predict))  test\_rmse = math.sqrt(mean\_squared\_error(testY, test\_predict))  # Print RMSE  print('Train RMSE: %.3f RMSE' % (train\_rmse))  print('Test RMSE: %.3f RMSE' % (test\_rmse))    # Plot the result  def plot\_result(trainY, testY, train\_predict, test\_predict):  actual = np.append(trainY, testY)  predictions = np.append(train\_predict, test\_predict)  rows = len(actual)  plt.figure(figsize=(15, 6), dpi=80)  plt.plot(range(rows), actual)  plt.plot(range(rows), predictions)  plt.axvline(x=len(trainY), color='r')  plt.legend(['Actual', 'Predictions'])  plt.xlabel('Observation number after given time steps')  plt.ylabel('Sunspots scaled')  plt.title('Actual and Predicted Values. The Red Line Separates The Training And Test Examples')    a = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv'  time\_steps = 12  train\_data, test\_data, data = get\_train\_test(a)  trainX, trainY = get\_XY(train\_data, time\_steps)  testX, testY = get\_XY(test\_data, time\_steps)    # Create model and train  model = create\_RNN(hidden\_units=3, dense\_units=1, input\_shape=(time\_steps,1),  activation=['tanh', 'tanh'])  model.fit(trainX, trainY, epochs=20, batch\_size=1, verbose=2)    # make predictions  train\_predict = model.predict(trainX)  test\_predict = model.predict(testX)    # Print error  print\_error(trainY, testY, train\_predict, test\_predict)    #Plot result  plot\_result(trainY, testY, train\_predict, test\_predict) |

Output:-



Experiment-9

Aim:- Creating NN From Scratch(Project)

Note:- This code is basically implementing a simple neural network from scratch using NumPy, Pandas and sklearn. In this code I have used binary classification. In which we trained and evaluate dataset from a CSV file, So this CSV file is belong to YouTube India trending Section, to predict watch video will get trend or not in the form of 0 and 1.

So, for that we have evaluate this code using Mean Square Error, Mean Absolute Error, Root Mean Square Root and accuracy.

So, if we breakdown our Code it include:-

1)Neural Network Class.

2)Sigmoid Activation Function.

3)Forward Pass

4)Backpropagation

5)Training of the data

6)Predictions

7)Evaluating process including Accuracy

8)Import CSV files and preprocessing it

9)Splitting of data in the form of train and test

10)Calling of every functions.

Code:-

|  |
| --- |
| import numpy as np  import pandas as pd  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler  #creating of neural Network  class NeuralNetwork:  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  #seting input, hidden and output layer  self.input\_size = input\_size  self.hidden\_size = hidden\_size  self.output\_size = output\_size  #randomly selecting input hidden and hidden output weights  self.weights\_input\_hidden = np.random.randn(self.input\_size, self.hidden\_size)  self.bias\_hidden = np.zeros((1, self.hidden\_size))  self.weights\_hidden\_output = np.random.randn(self.hidden\_size, self.output\_size)  self.bias\_output = np.zeros((1, self.output\_size))  #used in backpropogation  self.hidden\_layer\_output = None  #Use in epochs for storingthe loss data  self.losses = []  #Using activation in forword  def sigmoid(self, x):  return 1 / (1 + np.exp(-x))  #used function in backpropogation  def sigmoid\_derivative(self, x):  return x \* (1 - x)  #implementing of forword bias here  def forward(self, inputs):  #finding input to hidden layer here  hidden\_layer\_input = np.dot(inputs, self.weights\_input\_hidden) + self.bias\_hidden  self.hidden\_layer\_output = self.sigmoid(hidden\_layer\_input)  # hidden to output layer or final layer  output\_layer\_input = np.dot(self.hidden\_layer\_output, self.weights\_hidden\_output) + self.bias\_output  final\_output = self.sigmoid(output\_layer\_input)  return final\_output  #implementing of backward Propogation  def backward(self, inputs, targets, output, learning\_rate):  #finding error in hidden layer target-output  error = targets - output  #delta error part in backword  output\_delta = error \* self.sigmoid\_derivative(output)  hidden\_layer\_error = output\_delta.dot(self.weights\_hidden\_output.T)  hidden\_layer\_delta = hidden\_layer\_error \* self.sigmoid\_derivative(self.hidden\_layer\_output)  self.weights\_hidden\_output += self.hidden\_layer\_output.T.dot(output\_delta) \* learning\_rate  self.bias\_output += np.sum(output\_delta, axis=0, keepdims=True) \* learning\_rate  self.weights\_input\_hidden += inputs.T.dot(hidden\_layer\_delta) \* learning\_rate  self.bias\_hidden += np.sum(hidden\_layer\_delta, axis=0, keepdims=True) \* learning\_rate  #use to run the epochs and to find the loss of data  def train(self, inputs, targets, epochs, learning\_rate):  for epoch in range(epochs):  #it triggering forward  output = self.forward(inputs)  #it is basically triggering or calling the backward propogation  self.backward(inputs, targets, output, learning\_rate)  #here we are mean squaring the error and appending the loss  loss = np.mean(0.5 \* (targets - output) \*\* 2)  self.losses.append(loss)  #After which operations we want to see epoch  if epoch % 10 == 0:  print(f"Epoch {epoch}, Loss: {loss}")  def predict(self, inputs):  return self.forward(inputs)  #here we are finding the mean square error, mean absolute error and root mea square error.  def evaluate(self, predictions, targets):  mse = np.mean((targets - predictions) \*\* 2)  mae = np.mean(np.abs(targets - predictions))  rmse = np.sqrt(mse)  return mse, mae, rmse  #accuracy finding task is implementing here in this we are taking data in the form of binary data  def accuracy(self, predictions, targets, threshold=0.5):  binary\_predictions = (predictions > threshold).astype(int)  accuracy = np.mean(binary\_predictions == targets.reshape(-1, 1))  return accuracy  # Reading CSV file for the data Prediction  data = pd.read\_csv('India\_Youtube.csv')  #Preprocessing Process ib which data is in the form of binary digit means likes are is binary if its 1 then trending or if 0 then not trending.  threshold\_likes = 10000  data['trending'] = (data['likes'] > threshold\_likes).astype(int)  #so features we used for data store is views\_count, comment\_count, categoryID and taking target value as trending.  #mainly it is a Pandas function so, that it can particularly work on selected dataset.  features = data[['view\_count', 'comment\_count', 'categoryId']]  target = data['trending']  #It is sk learn part which basically used for mean and standar deviation in data where mean is 0 and standar deviation is 1.  scaler = StandardScaler()  features\_scaled = scaler.fit\_transform(features) #transform is for computation mean and standard deviation in dataonly  #here data is splitting in train and test  X\_train, X\_test, y\_train, y\_test = train\_test\_split(features\_scaled, target, test\_size=0.2, random\_state=42)  #self defined parameter used in NN  input\_size = X\_train.shape[1]  hidden\_size = 8  output\_size = 1  learning\_rate = 0.01  #Calling NeuralNetwork  nn = NeuralNetwork(input\_size, hidden\_size, output\_size)  #calling training model  nn.train(X\_train, y\_train.values.reshape(-1, 1), epochs=500, learning\_rate=learning\_rate)  #calling Predictions testing  predictions = nn.predict(X\_test)  #calling evaluation function like mean square error, mean absolute error and root mean square error and Accuracy part  mse, mae, rmse = nn.evaluate(predictions, y\_test.values.reshape(-1, 1))  accuracy = nn.accuracy(predictions, y\_test.values.reshape(-1, 1))  #Output we are showing  print(f"Mean Squared Error: {mse:.4f}")  print(f"Mean Absolute Error: {mae:.4f}")  print(f"Root Mean Squared Error: {rmse:.4f}")  print(f"Accuracy on the test set: {accuracy \* 100:.2f}%") |

Output:-

