

*NAME: AYUSH CHANCHAL*

*SAP ID: 500097569*

*BATCH: B6*

*SUBJECT: NEURAL NETWORK LAB FILE*

*FACULTY: SUGANDHA SHARMA*

Ayush Chanchal

Sap:500097569

Batch:B6

EXPERIMENT:01

AIM: Simple Neural Network:

A simple neural network, also known as a feedforward neural network, consists of three main types of layers: an input layer, one or more hidden layers, and an output layer.

Steps:

Input Layer:

The input layer consists of nodes (neurons) that represent the features of the input data. Each node corresponds to one feature.

Hidden Layer:

The hidden layer(s) are between the input and output layers. They perform computations on the input data and pass the results to the output layer. Each node in the hidden layer is connected to every node in the previous layer.

Output Layer:

The output layer produces the final result of the neural network’s computation. The number of nodes in the output layer depends on the nature of the problem (e.g., binary classification, multi-class classification, regression, etc.).

Connections (Weights):

Each connection between nodes has an associated weight. These weights are learned during the training process and represent the strength of the connection.

Activation Functions:

Each node (except the input nodes) has an activation function that applies a non-linear transformation to the weighted sum of its inputs. Common activation functions include sigmoid, tanh, ReLU, etc.

Bias:

Each node (except the input nodes) also has a bias term. The bias helps in shifting the activation function and allows the network to learn more complex relationships.

Forward Pass:

During the forward pass, the input data is fed through the network, and computations are performed layer by layer until the output is obtained.

Backpropagation:

After the forward pass, the network’s output is compared to the true target values. The error (the difference between the predicted and actual values) is then propagated backward through the network using an optimization algorithm (e.g., gradient descent) to update the weights and biases.

Training:

The process of adjusting the weights and biases based on the error is called training. This is done over multiple iterations (epochs) until the model’s performance improves.

Prediction:

Once the network is trained, it can be used to make predictions on new, unseen data by performing a forward pass with the new input.

Q). Implementation of Simple neural network using Activation Functions.

Sourcecode:

import numpy as np

class NeuralNetwork():

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

np.random.seed(1)

self.synaptic\_weights = 2\*np.random.random((3, 1))-1

def sigmoid(self, x):

return 1/(1+np.exp(-x))

def sigmoid\_derivative(self, x):

return x\*(1-x)

def train(self, training\_inputs, training\_outputs, training\_iterations):

for iteration in range(training\_iterations):

output = self.think(training\_inputs)

error = training\_outputs-output

adjustments = np.dot(training\_inputs.T, error \*

self.sigmoid\_derivative(output))

self.synaptic\_weights += adjustments

def think(self, inputs):

inputs = inputs.astype(float)

output = self.sigmoid(np.dot(inputs, self.synaptic\_weights))

return output

if \_\_name\_\_ == “\_\_main\_\_”:

neural\_network = NeuralNetwork()

print(“Beginning Randomly generated weights:”)

print(neural\_network.synaptic\_weights)

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

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)

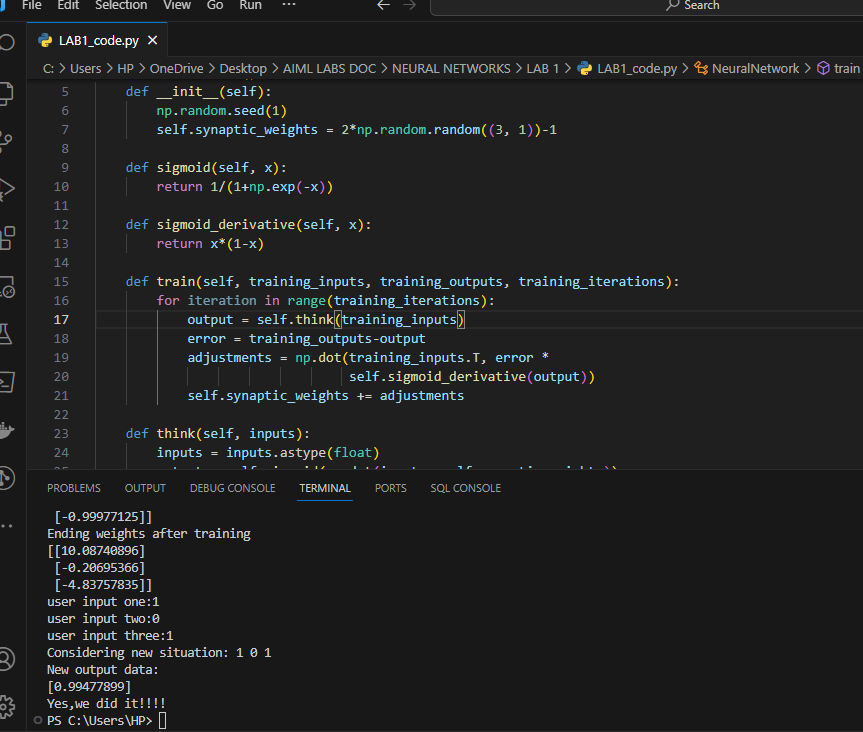
print(“New output data:”)

print(neural\_network.think(

np.array([user\_input\_one, user\_input\_two, user\_input\_three])))

print(‘Yes,we did it!!!!’)

Screenshot:



NAME: AYUSH CHANCHAL

SAP ID: 500097569

Batch: 06

EXPERIMENT:02

AIM: Activation function in Simple neural network

1. Explain the importance of Activation Function in Neural Network.

: - An activation function decides the input of a neuron to the network is important or not in the process of prediction while using mathematical operations. It is use to convert neurons in non-linear to predict the pattern and on the basis of that model will predict the output.

Types of Activation Function:

1. Sigmoid Function: In this type of activation function, we give the input any real number and we get the output in the range of 0 to 1.

Formula:

f(x)=1/(1+exp-x)

x=W0x1+w1x2+w2x3……..

1. Tanh Function (Hyperbolic Tangent): This Function is very similar to sigmoid function because it has the same S graph as the sigmoid function but it differs in the range. It take the real number as input but the output range is from -1 to +1.

Formula:

f(x) = (ex – e-x) / (ex + e-x)

x=W0x1+w1x2+w2x3……..

1. RELU Function (Rectified Linear Unit): It has a derivative function and allows for backpropagation while simultaneously making it computationally efficient. If the output is less than 0 then only the neurons will be deactivated.

Formula:

f(x): max (0, x)

x=W0x1+w1x2+w2x3……..

1. Leaky RELU Function: It is the modified version of RELU Function. Instead of defining the RELU activation function as 0 for negative values of inputs(x), we define it as an extremely small linear component of x.

Formula:

f(x)= max (0.01\*x, x)

x=W0x1+w1x2+w2x3……..

1. SoftMax Function: It is the last Activation function of neural network and it is use to normalize the output of a network to a probability distribution over predicted output class.

Formula:

S(y)i= exp(yi)/(∑j=1 exp(yj))

Where,

y= input value

yi= i-th element of the input vector

∑exp(yi)= a normalization value

1. Implementing the popular activation functions.

I). Sigmoid Function:

Source code:

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')

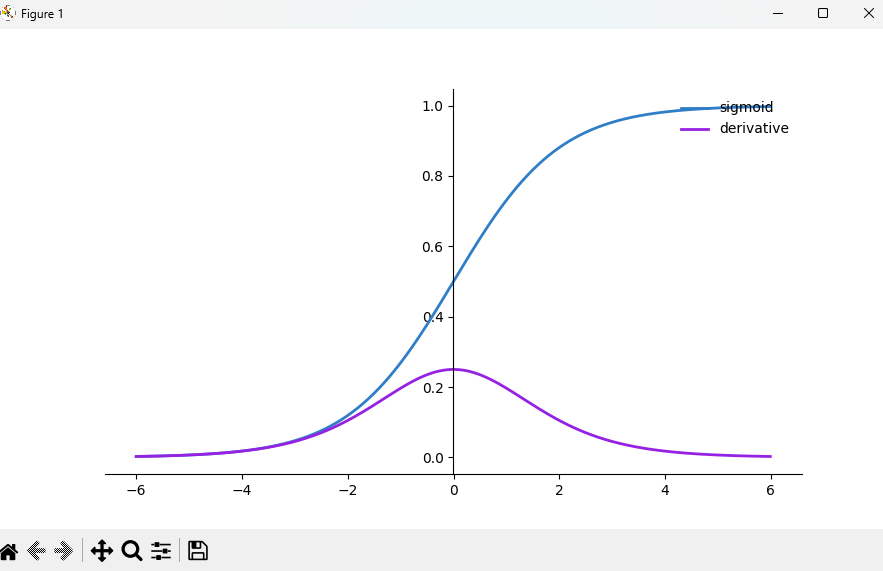
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=2, label="sigmoid")

ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=2, label="derivative")

ax.legend(loc="upper right", frameon=False)

plt.show()

Graph:



II). Tangent Hyperbolic Function

Source code:

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(-6,6,0.01)

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')

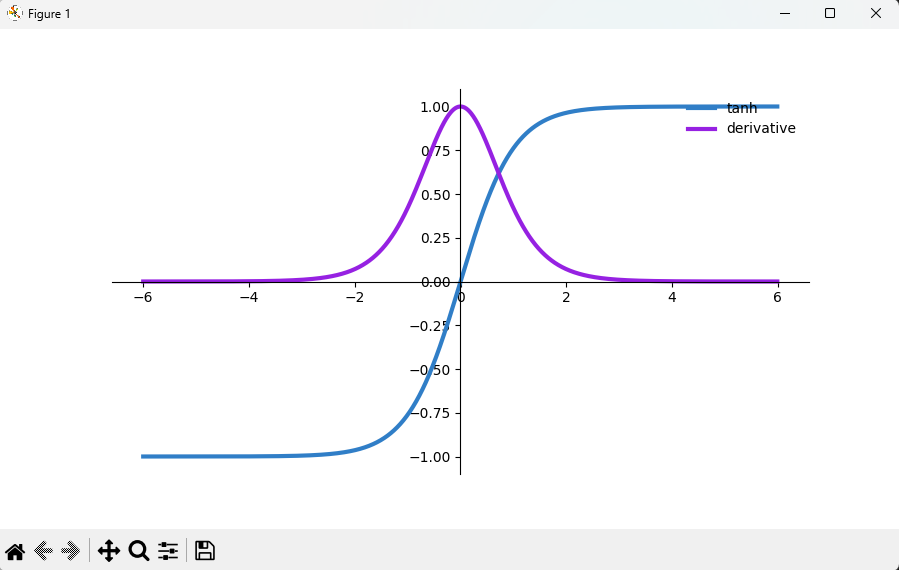
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")

ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")

ax.legend(loc="upper right", frameon=False)

plt.show()

Graph:



III). RELU Function

Source code:

import numpy as np

import matplotlib.pyplot as plt

def ReLU(x):

data = [max(0,value) for value in x]

return np.array(data, dtype=float)

def der\_ReLU(x):

data = [1 if value>0 else 0 for value in x]

return np.array(data, dtype=float)

x\_data = np.linspace(-10,10,100)

y\_data = ReLU(x\_data)

dy\_data = der\_ReLU(x\_data)

plt.plot(x\_data, y\_data, x\_data, dy\_data)

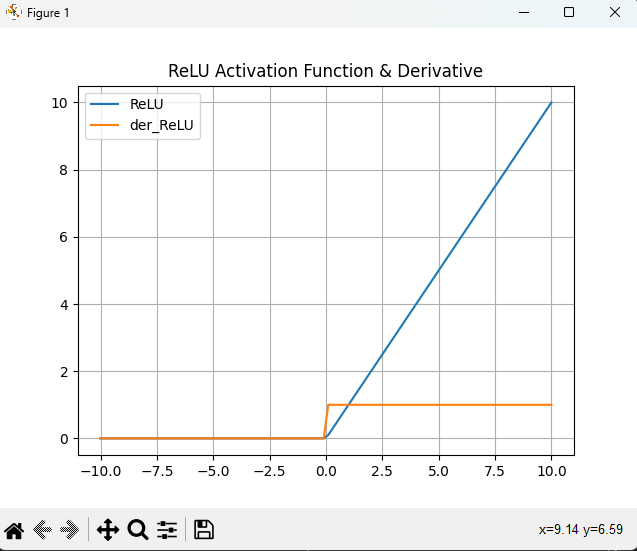
plt.title('ReLU Activation Function & Derivative')

plt.legend(['ReLU','der\_ReLU'])

plt.grid()

plt.show()

Graph:



IV). Leaky RELU Function

Source code:

import numpy as np

import matplotlib.pyplot as plt

def leaky\_ReLU(x):

data = [max(0.05\*value, value) for value in x]

return np.array(data, dtype=float)

def der\_leaky\_ReLU(x):

data = [1 if value > 0 else 0.05 for value in x]

return np.array(data, dtype=float)

x\_data = np.linspace(-10, 10, 100)

y\_data = leaky\_ReLU(x\_data)

dy\_data = der\_leaky\_ReLU(x\_data)

plt.plot(x\_data, y\_data, x\_data, dy\_data)

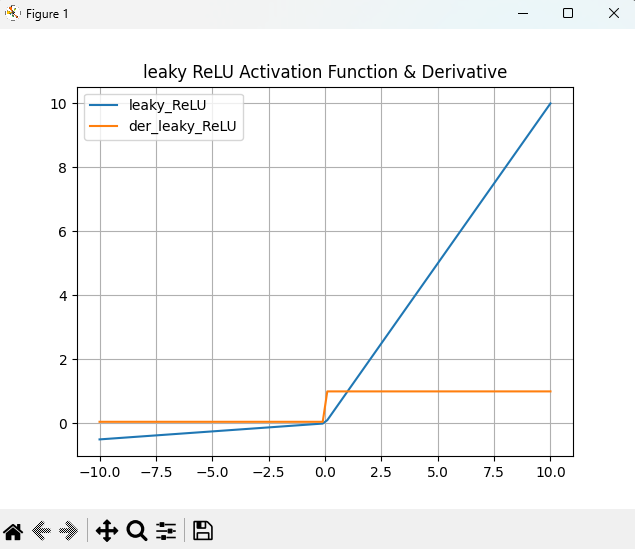
plt.title('leaky ReLU Activation Function & Derivative')

plt.legend(['leaky\_ReLU', 'der\_leaky\_ReLU'])

plt.grid()

plt.show()

Graph:



V). SoftMax Function

Source code:

import matplotlib.pyplot as plt

import numpy as np

def softmax(x):

''' Compute softmax values for each sets of scores in x. '''

return np.exp(x) / np.sum(np.exp(x), axis=0)

x = np.linspace(-10, 10)

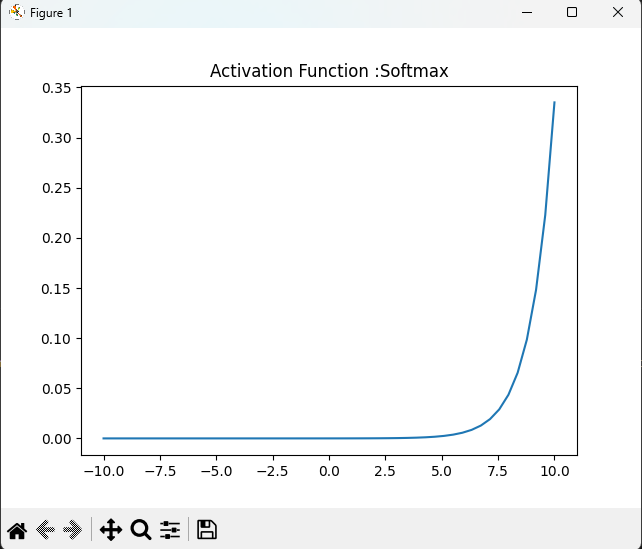
plt.plot(x, softmax(x))

plt.axis('tight')

plt.title('Activation Function :Softmax')

plt.show()

Graph:



1. Advantages and Disadvantages of the types of Activation Function.
2. Sigmoid function

Advantages:

* Output values range between 0 and 1, mimicking a probability distribution.
* It is differentiable, making it suitable for gradient-based optimization.

Disadvantages:

* Prone to the vanishing gradient problem, which can slow down or halt learning in deep networks.
* Outputs are not zero-centered, which may hinder learning

1. Hyperbolic Tangent (tanh) Activation Function:

Advantages:

* Similar to the sigmoid function, but zero-centered, which can speed up convergence.
* It is differentiable.

Disadvantages:

* Still susceptible to the vanishing gradient problem.
* The output is bounded between -1 and 1, which may not be appropriate for all scenarios.

1. Rectified Linear Unit (ReLU):

Advantages:

* Computationally efficient due to its simple thresholding.
* Helps mitigate the vanishing gradient problem.
* Promotes sparse activations, making the network easier to optimize.

Disadvantages:

* Can suffer from the "dying ReLU" problem, where neurons get stuck and cease to learn.

1. Leaky ReLU:

Advantages:

* Addresses the "dying ReLU" problem by allowing a small gradient for negative inputs.
* Helps with the vanishing gradient problem.

Disadvantages:

Adds complexity with an extra hyperparameter (slope of the negative part).

Name: Ayush Chanchal

Sap:500097569

Batch: B6

EXPERIMENT: 03

Aim: Implementation of Bias in neural network

1. Explain the concept of bias in neural networks?

: Bias is a constant help the model to best fit on the given data. It is like the intercept which added to the linear equation. It is an additional parameter which is use to adjust the output along with the weighted summation of the inputs in neural network. The bias is a crucial parameter in neural networks that helps in fine-tuning the activation and output of each neuron. It contributes to the network's ability to learn complex relationships and make accurate predictions.

Formula:

Y=∑(wi \* xi) + B

Where, w=connection weights

B=Bias

x=inputs

1. Implementation of Backpropagation in neural network?

:

Source code:

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

import matplotlib.pyplot as plt

data=load\_iris()

x=data.data

y=data.target

y=pd.get\_dummies(y).values

print(y[:3])

x\_train,x\_test,y\_train,y\_test=train\_test\_split(x,y,test\_size=20,random\_state=4)

learning\_rate=0.1

iterations=5000

n=y\_train.size

input\_size=4

hidden\_size=2

output\_size=3

results=pd.DataFrame(columns=["mse","accuracy"])

np.random.seed(10)

w1=np.random.normal(scale=0.5,size=(input\_size,hidden\_size))

w2=np.random.normal(scale=0.5,size=(hidden\_size,output\_size))

def sigmoid(x):

return 1/(1+np.exp(-x))

def mean\_squared\_error(y\_pred,y\_true):

return ((y\_pred-y\_true)\*\*2).sum()/(2\*y\_pred.size)

def accuracy(y\_pred,y\_true):

acc=y\_pred.argmax(axis=1)==y\_true.argmax(axis=1)

return acc.mean()

for itr in range(iterations):

z1=np.dot(x\_train,w1)

a1=sigmoid(z1)

z2=np.dot(a1,w2)

a2=sigmoid(z2)

mse=mean\_squared\_error(a2,y\_train)

acc=accuracy(a2,y\_train)

results=results.\_append({"mse":mse,"accuracy":acc},ignore\_index=True)

e1=a2-y\_train

dw1=e1\*a2\*(1-a2)

e2=np.dot(dw1,w2.T)

dw2=e2\*a1\*(1-a1)

w2\_update=np.dot(a1.T,dw1)/n

w1\_update=np.dot(x\_train.T,dw2)/n

w2=w2-learning\_rate\*w2\_update

w1=w1-learning\_rate\*w1\_update

results.mse.plot(title="Mean Squared Error")

plt.show()

results.accuracy.plot(title="Accuracy")

plt.show()

# Testing

z1=np.dot(x\_test,w1)

a1=sigmoid(z1)

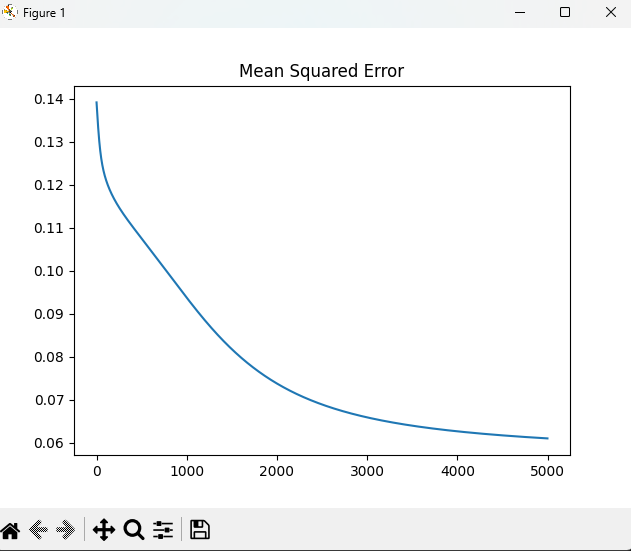
z2=np.dot(a1,w2)

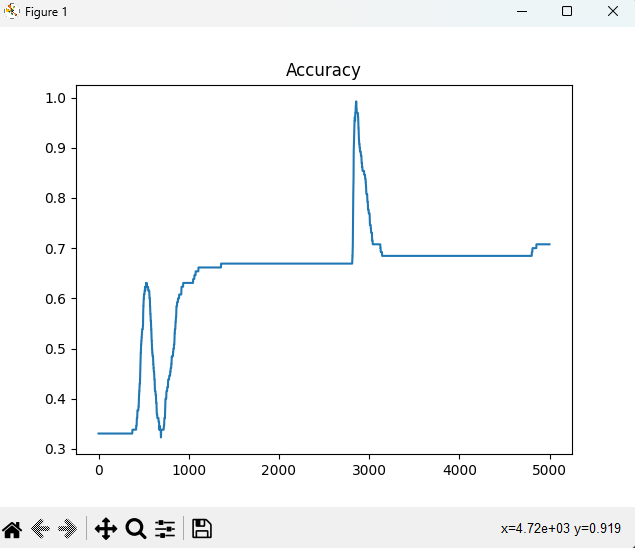
a2=sigmoid(z2)

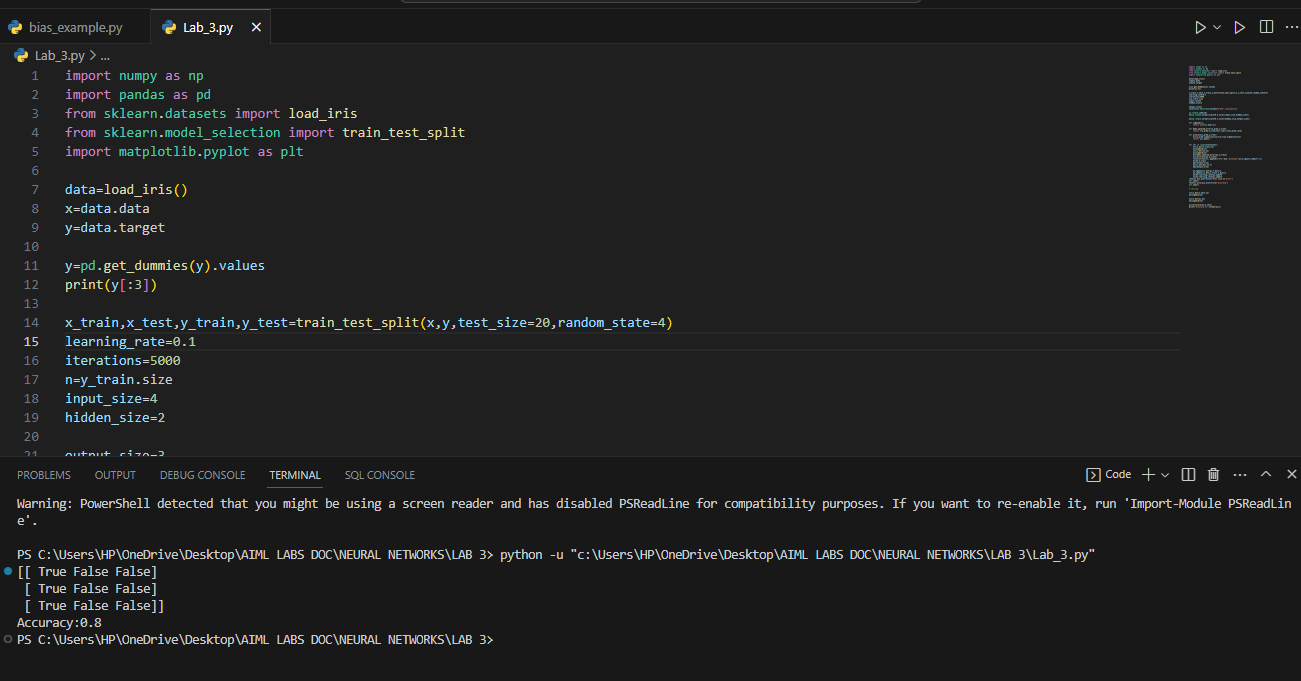
acc=accuracy(a2,y\_test)

print("Accuracy:{}".format(acc))

Screenshot:







Ayush Chanchal

Sap:500097569

Batch: B6

EXPERIMENT: 04

Aim: Implementation of Hebbian learning rule

Q). Explain the Hebbian Learning Rule. Implement the same in Python.

: - Hebbian Learning- Hebbian learning is an important idea in both brains and artificial neural networks. It explains how when two neurons are active at the same time, their connection gets stronger. This is like how we remember things in our brains. It's also used in making computer models that learn. So, Hebbian learning helps us understand how our brains learn and remember, and it's also useful for making smart computer programs.

Source code:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# dataset = pd.read\_csv('IRIS.csv')

# print(dataset.head())

num\_inputs = int(input("Enter the number of inputs: "))

num\_samples = int(input("Enter the number of training samples: "))

train\_set = [[int(input()) for x in range(num\_inputs)]

for y in range(num\_samples)]

C = float(input("Enter the value of learning constant: "))

weights = [1, -1]

def sign\_function(input\_value):

return (1 if input\_value >= 0 else -1)

for iteration in range(len(train\_set)):

net\_value = 0

for i in range(len(weights)):

net\_value += weights[i] \* train\_set[iteration][i]

signed\_value = sign\_function(net\_value)

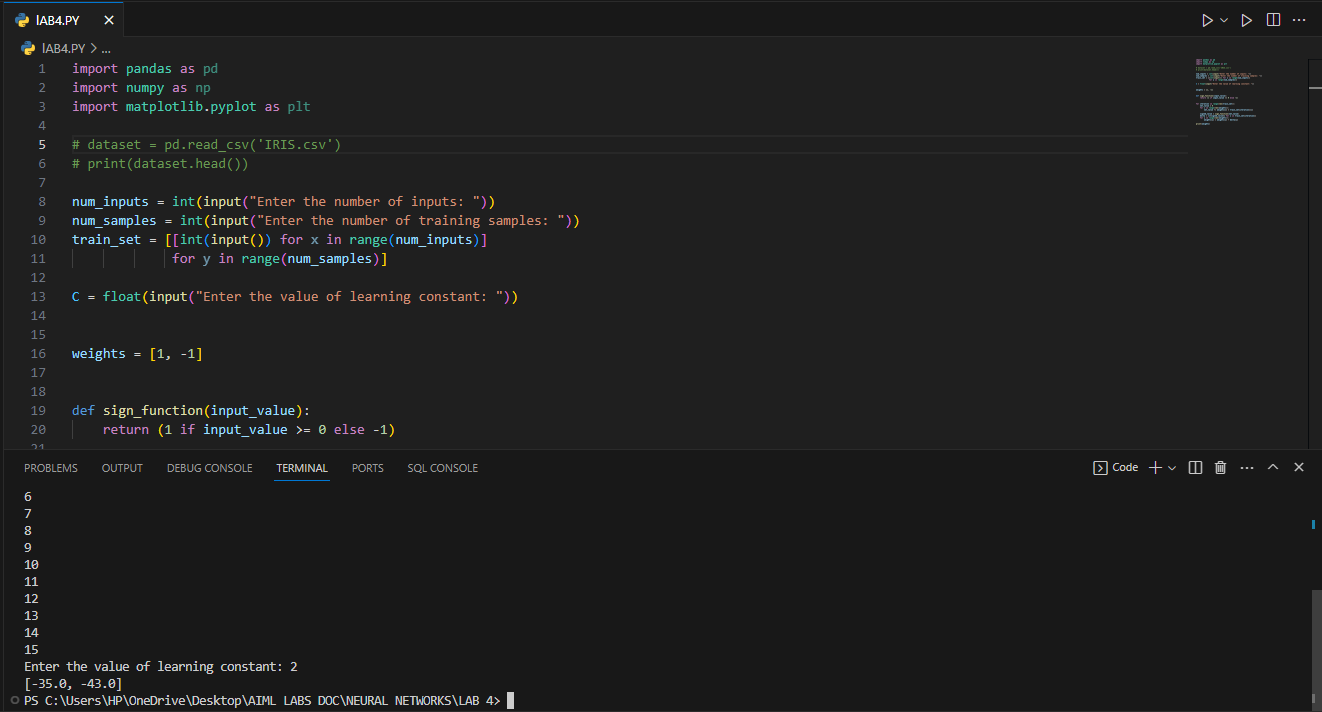
delta = [C\*signed\_value\*x for x in train\_set[iteration]]

for j in range(len(weights)):

weights[j] = weights[j] + delta[j]

print(weights)

Screenshot:



Ayush Chanchal

Sap:500097569

Batch: B6

EXPERIMENT:05

Aim : competitive learning implementation

Description:

Observing how learning is achieved in the MNIST handwritten digits dataset involves training a neural network on this dataset and analyzing its performance over time.

Load the MNIST Dataset:

Download the MNIST dataset. It consists of a training set and a test set of handwritten digits.

Each image is 28x28 pixels, and each pixel value is between 0 and 255, representing grayscale intensity.

Preprocess the Data:

Normalize the pixel values to be between 0 and 1. This helps in faster convergence during training.

Design a Neural Network:

Create a neural network architecture suitable for image classification. A simple feedforward neural network or a convolutional neural network (CNN) is commonly used for this task.

Compile the Model:

Define the loss function (typically categorical cross-entropy for classification tasks) and the optimizer (e.g., stochastic gradient descent).

Train the Model:

Train the model using the training data. This involves forward pass (predicting) and backward pass (backpropagating the error) through the network.

Evaluate the Model:

Use the test set to evaluate the model's performance. Look at metrics like accuracy, precision, recall, etc.

Visualize the Learning Curve:

Plot the training and validation loss and accuracy over epochs. This shows how the model's performance changes over time.

Inspect Misclassified Samples:

Look at some of the samples the model got wrong. This can give insights into what types of images are challenging for the model.

Interpretation:

Analyze what the network is learning. For example, in convolutional layers, you can look at the learned filters.

Experiment:

Try different network architectures, regularization techniques, or even different deep learning frameworks to see how they affect learning.

Q). Implement the competitive learning algorithm using a simple neural network and train it using data from the MNIST Handwritten digits dataset, observing how learning is achieved.

Source code:

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

train\_df=pd.read\_csv("mnist\_train.csv")

train\_image=dict()

for id,row in train\_df.iterrows():

x=row[1:785].to\_numpy()

train\_image[id]={'label': row[0],'data' : row[1:785].to\_numpy() / np.sqrt(np.dot(x, x))}

def normalize(vec):

return vec / np.sqrt(vec.dot(vec))

alpha= 0.20

iterations = 100000

n\_output = 10

n\_pixels = 784

n\_samples = len(train\_image)

rng = np.random.default\_rng()

w\_chg\_ls = list()

W = np.random.rand(n\_output, n\_pixels)

for x in range(n\_output):

W[x] = normalize(W[x])

def training():

for t in range(iterations):

if t % 1000:

w\_chg\_ls.append(np.copy(W))

rand\_i = rng.integers(n\_samples)

input\_vec=train\_image[rand\_i]['data']

win\_index=np.argmax(np.dot(W,input\_vec))

W[win\_index]+=alpha\*(input\_vec-W[win\_index])

return W

weights = training()

def visualization():

fig, ax = plt.subplots(nrows=10, ncols=5, figsize=(15, 15))

for x in range(10):

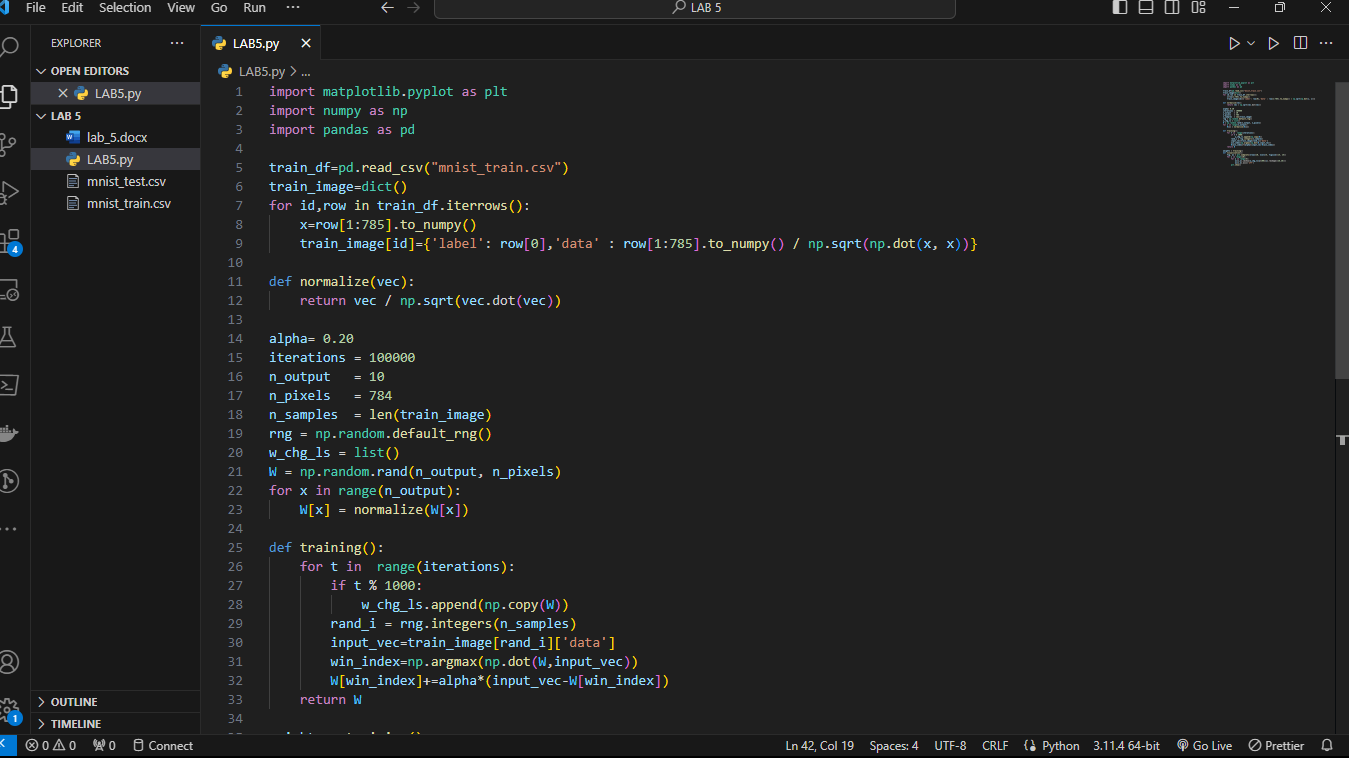
for w in range(5):

ax[x,w].imshow(w\_chg\_ls[w\*100][x].reshape((28,28)))

ax[x,w].axis('off')

plt.show()

ScreenShot:



Ayush Chanchal

Sap:500097569

Batch: B6

EXPERIMENT:06

Aim: Implement Backpropagation

**Backpropagation:**

Backpropagation is a method used to train neural networks. It's like a feedback mechanism that helps the network learn from its mistakes. Here's a simple explanation:

* Forward Pass: During the training process, the neural network makes predictions on the input data. These predictions might not be very accurate at first.
* Calculate Error: We compare these predictions with the actual desired outputs to see how wrong or right the network's predictions are.
* Backward Pass (Backpropagation): This is where the magic happens. We take the error and move it backward through the network, kind of like telling each neuron how much it contributed to the error.
* Adjust Weights: Neurons have weights that determine how much they influence the output. During backpropagation, these weights get adjusted based on how responsible they were for the error. Neurons that contributed more to the error get bigger adjustments.
* Repeat: Steps 1-4 are repeated for many iterations (epochs) until the network's predictions become accurate.

**Implement Backpropagation with Error Correction learning in Python.**

**Source Code:**

import numpy as np

def initialize\_parameter(layers):

parameters={}

for i in range(1,len(layers)):

parameters["W"+str(i)]=np.ones((layers[i-1],layers[i]))\*0.1

return parameters

def forward\_prop(X,parameters):

A=X

for i in range(1,len(parameters)+1):

A\_prev=A

W1=parameters["W"+str(i)]

# print(f"A{i}={A\_prev}")

# print(f"W{i}={W1}")

A=np.dot(A,W1)

# print(f"output={A}")

return A,A\_prev

def update\_parameter(X,layers,learning\_rate,Y):

parameters=initialize\_parameter(layers)

for epochs in range(20):

for i in range(X.shape[0]):

Y\_hat,A=forward\_prop(X[i],parameters)

for row1 in range(parameters["W2"].shape[0]):

for col1 in range(parameters["W2"].shape[1]):

parameters["W2"][row1][col1]=parameters["W2"][row1][col1]+(learning\_rate\*(Y[i]-Y\_hat)\*2\*[row1])

for row in range(parameters["W1"].shape[0]):

for col in range(parameters["W1"].shape[1]):

parameters["W1"][row][col]=parameters["W1"][row][col]+learning\_rate\*(Y[i]-Y\_hat)\*2\*parameters["W2"][col][0]\*X[i][row]

return(parameters)

X\_train=np.array([[1,2,3],[4,5,6]])

Y\_train=np.array([[1],[4]])

layers=[3,2,1]

weight\_best=update\_parameter(X\_train,layers,0.01,Y\_train)

x\_test=np.array([[7,8,9],[10,11,12]])

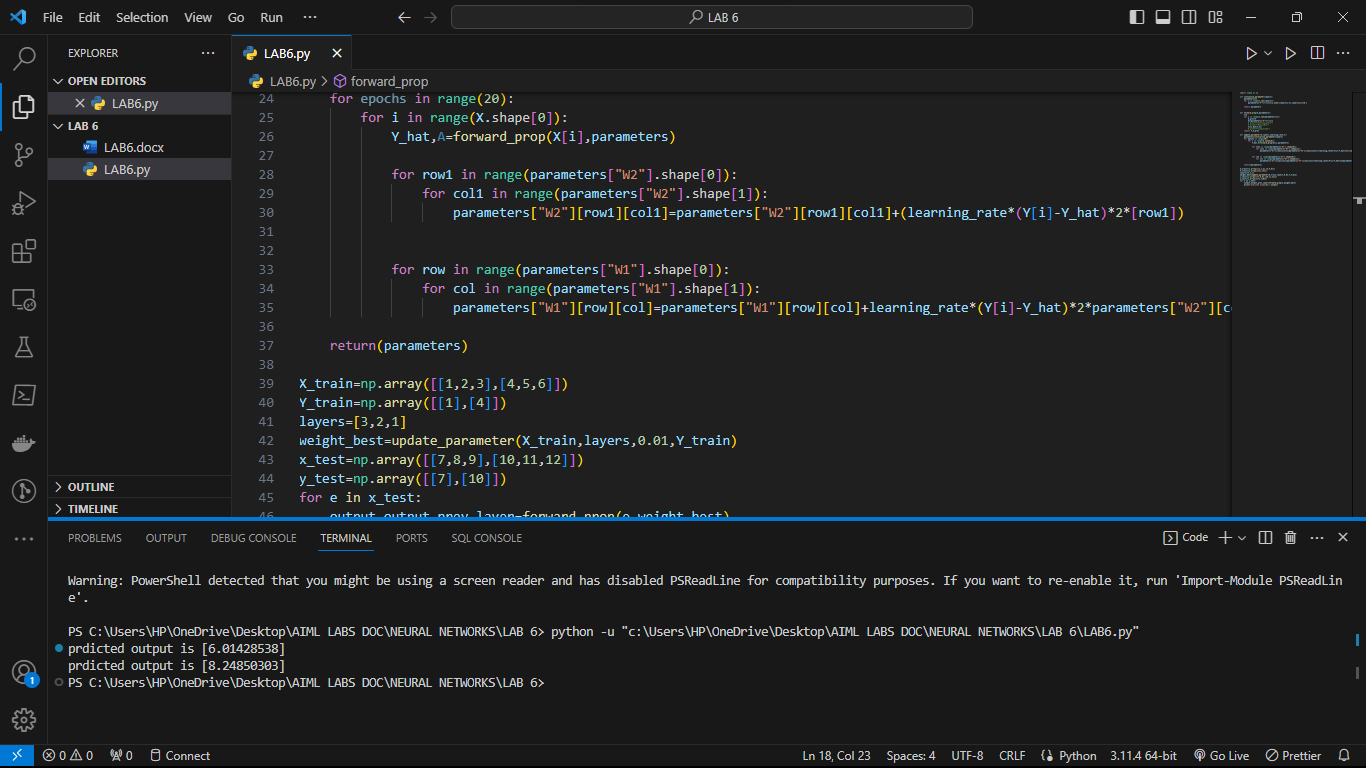
y\_test=np.array([[7],[10]])

for e in x\_test:

output,output\_prev\_layer=forward\_prop(e,weight\_best)

print("prdicted output is",output)

Screenshot:



Name: Ayush Chanchal

Sap id: 500097569

Batch: B6

*EXPERIMENT: 7*

Aim: IMPLEMENTATION OF CNN

Q). WHAT IS CNN?

CNN stands for Convolutional Neural Network. It is a type of artificial neural network used in machine learning, particularly well-suited for tasks related to image and video recognition, as well as other grid-like data. CNNs are designed to automatically and adaptively learn patterns from data through a process known as convolution.

Here are some key components and concepts associated with CNNs:

1. Convolution: In a CNN, convolution is the core operation. It involves passing a small filter (also called a kernel) over the input data to extract local patterns. These patterns can represent features like edges, textures, or more complex structures. Convolutional layers help reduce the number of parameters and allow the network to focus on local patterns.

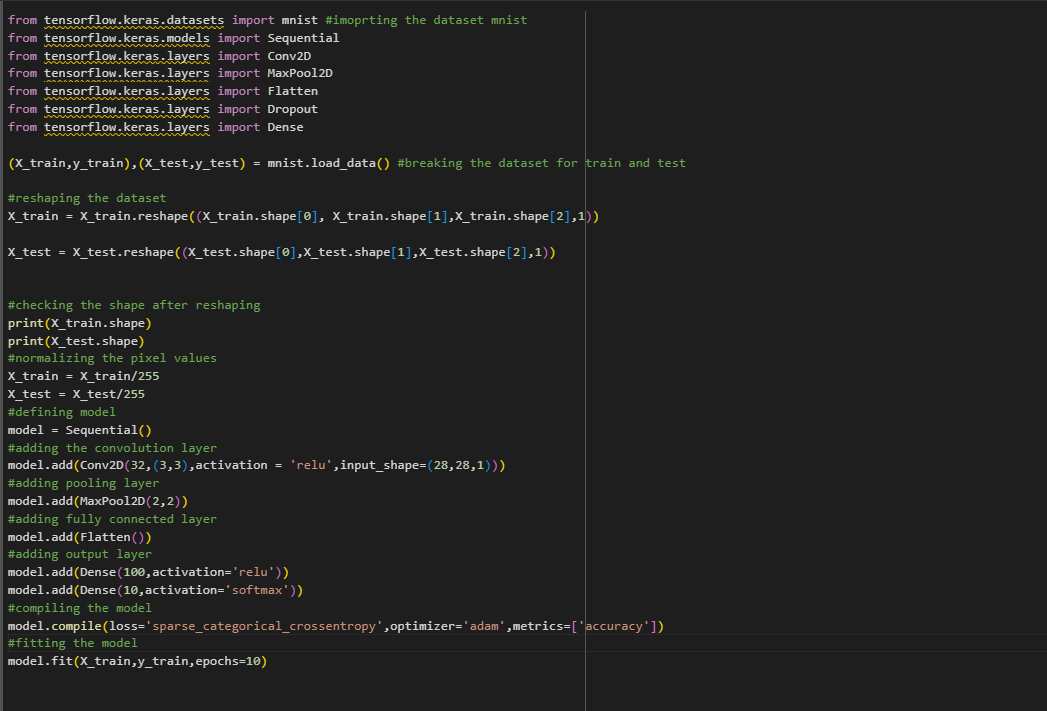
2. Pooling: Pooling layers are used to downsample the spatial dimensions of the data. The most common type of pooling is max pooling, which retains the maximum value from a small region of the input. Pooling helps reduce the computational burden and makes the network more robust to variations in input.

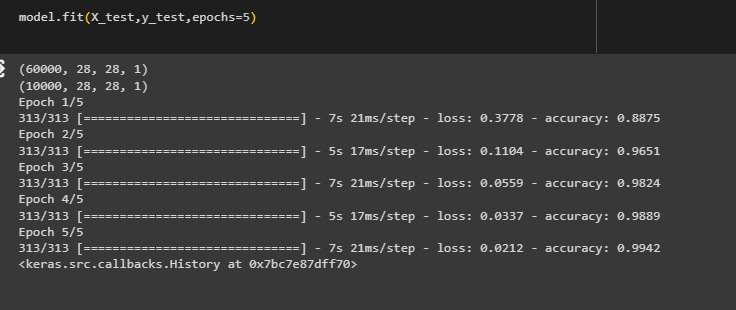
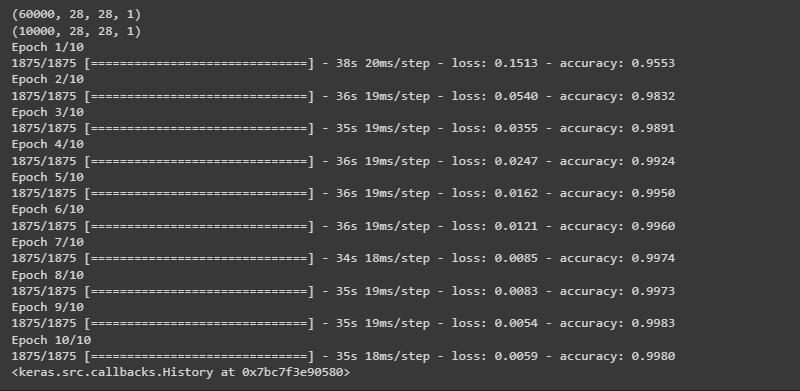
3. Convolutional Layers: These layers consist of multiple convolution operations, followed by activation functions, which introduce non-linearity into the network. Convolutional layers extract hierarchical features from the input data.

4. Fully Connected Layers: After several convolutional and pooling layers, CNNs often have one or more fully connected layers that are similar to traditional neural networks. These layers help in making predictions or classifications based on the features learned in the earlier layers.

5. Activation Functions: Common activation functions used in CNNs include ReLU (Rectified Linear Unit) and its variants. These functions introduce non-linearity and allow the network to learn complex patterns.

CNNs are widely used in tasks such as image classification, object detection, facial recognition, and even in more complex applications like natural language processing when dealing with sequential data (using methods like 1D convolutions). They have revolutionized the field of computer vision and are a fundamental technology in deep learning.

IMPLEMENTATION:-



Name: Ayush Chanchal

Sap: 500097569

Batch: B6

EXPERIMENT: 08

Aim: Implementation And, or, nand on perceptron

**Perceptron for the and function:**

In this we will program a Neural Network in Python which implements the logical "And" function. It is defined for two inputs in the following way:

| **Input1** | **Input2** | **Output** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

As we know that a neural network with one perceptron and two input values can be interpreted as a decision boundary, i.e. straight line dividing two classes. The two classes we want to classify in our example look like this:

import matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

xmin, xmax = -0.2, 1.4

X = np.arange(xmin, xmax, 0.1)

ax.scatter(0, 0, *color*="r")

ax.scatter(0, 1, *color*="r")

ax.scatter(1, 0, *color*="r")

ax.scatter(1, 1, *color*="g")

ax.set\_xlim([xmin, xmax])

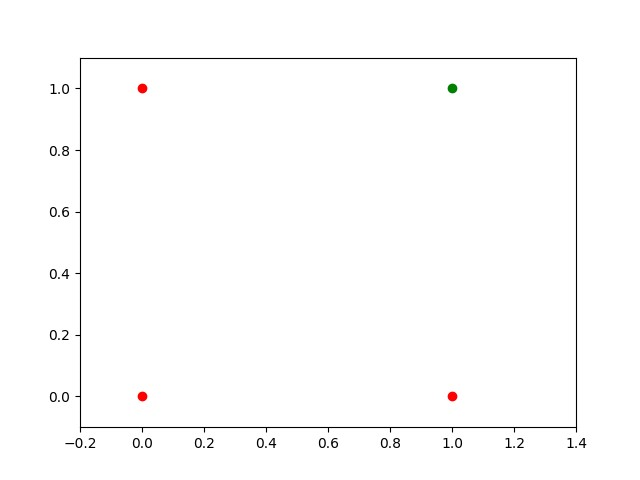
ax.set\_ylim([-0.1, 1.1])

m = -1

#ax.plot(X, m \* X + 1.2, label="decision boundary")

plt.plot()

plt.show()



We found out that such a primitive neural network is only capable of creating straight lines going through the origin.

So dividing line like this:

import matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

xmin, xmax = -0.2, 1.4

X = np.arange(xmin, xmax, 0.1)

ax.set\_xlim([xmin, xmax])

ax.set\_ylim([-0.1, 1.1])

m = -1

for m in np.arange(0, 6, 0.1):

    ax.plot(X, m \* X )

ax.scatter(0, 0, *color*="r")

ax.scatter(0, 1, *color*="r")

ax.scatter(1, 0, *color*="r")

ax.scatter(1, 1, *color*="g")

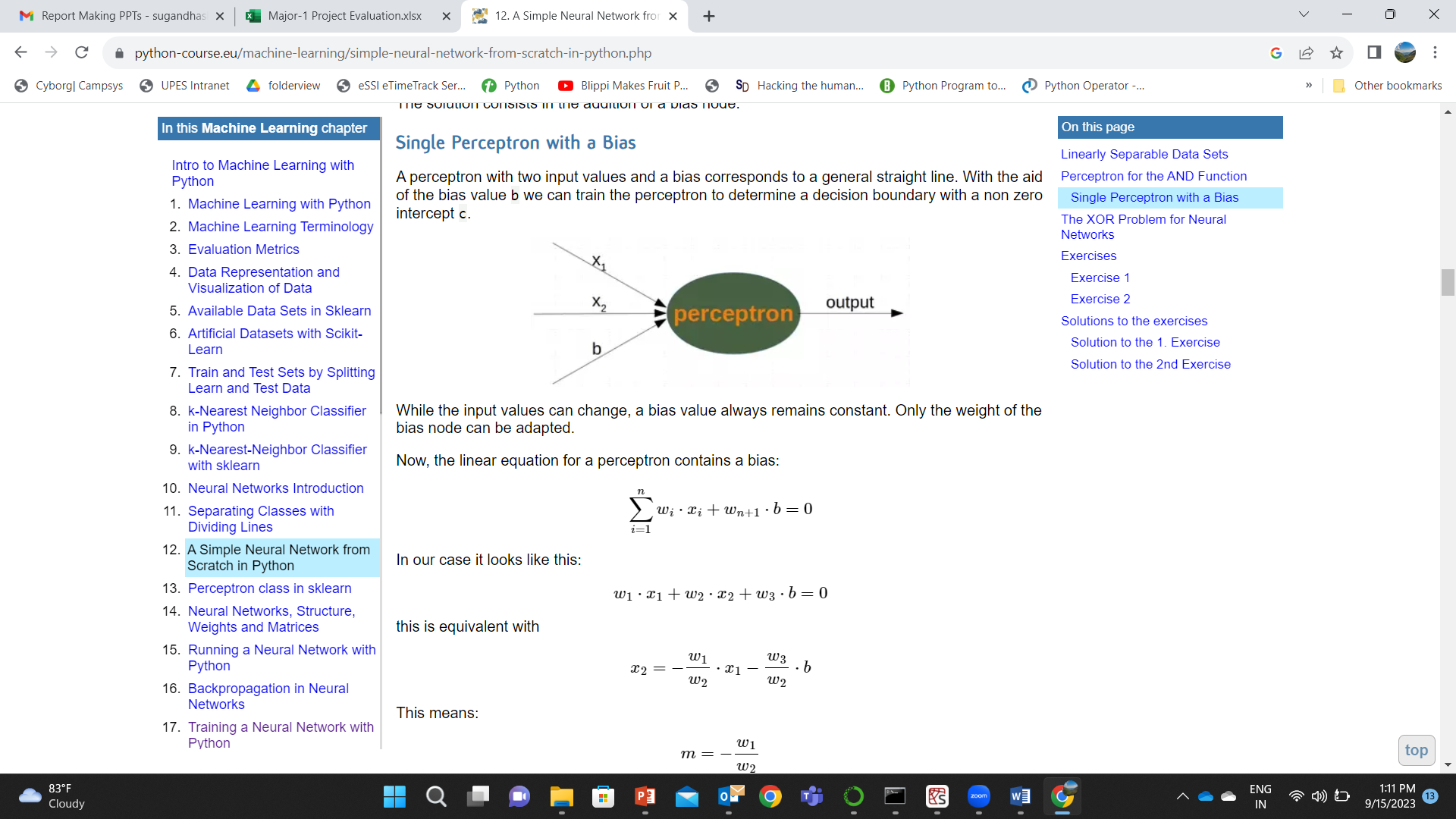
plt.plot()

plt.show()

### 

### **Single Perceptron with a Bias**

A perceptron with two input values and a bias corresponds to a general straight line. With the aid of the bias value b we can train the perceptron to determine a decision boundary with a non zero intercept c.



import numpy as np

from collections import Counter

class Perceptron:

    def \_\_init\_\_(*self*,

*weights*,

*bias*=1,

*learning\_rate*=0.3):

        """

        'weights' can be a numpy array, list or a tuple with the

        actual values of the weights. The number of input values

        is indirectly defined by the length of 'weights'

        """

*self*.weights = np.array(*weights*)

*self*.bias = *bias*

*self*.learning\_rate = *learning\_rate*

    @staticmethod

    def unit\_step\_function(*x*):

        if  *x* <= 0:

            return 0

        else:

            return 1

    def \_\_call\_\_(*self*, *in\_data*):

*in\_data* = np.concatenate( (*in\_data*, [*self*.bias]) )

        result = *self*.weights @ *in\_data*

        return Perceptron.unit\_step\_function(result)

    def adjust(*self*,

*target\_result*,

*in\_data*):

        if type(*in\_data*) != np.ndarray:

*in\_data* = np.array(*in\_data*)  #

        calculated\_result = *self*(*in\_data*)

        error = *target\_result* - calculated\_result

        if error != 0:

*in\_data* = np.concatenate( (*in\_data*, [*self*.bias]) )

            correction = error \* *in\_data* \* *self*.learning\_rate

*self*.weights += correction

    def evaluate(*self*, *data*, *labels*):

        evaluation = Counter()

        for sample, label in zip(*data*, *labels*):

            result = *self*(sample) # predict

            if result == label:

                evaluation["correct"] += 1

            else:

                evaluation["wrong"] += 1

        return evaluation

import numpy as np

from perceptron import \*

def labelled\_samples(*n*):

    for \_ in range(*n*):

        s = np.random.randint(0, 2, (2,))

        yield (s, 1) if s[0] == 1 and s[1] == 1 else (s, 0)

p = Perceptron(*weights*=[0.3, 0.3, 0.3],

*learning\_rate*=0.2)

for in\_data, label in labelled\_samples(30):

    p.adjust(label,

             in\_data)

test\_data, test\_labels = list(zip(\*labelled\_samples(30)))

evaluation = p.evaluate(test\_data, test\_labels)

print(evaluation)

mport matplotlib.pyplot as plt

import numpy as np

fig, ax = plt.subplots()

xmin, xmax = -0.2, 1.4

X = np.arange(xmin, xmax, 0.1)

ax.scatter(0, 0, *color*="r")

ax.scatter(0, 1, *color*="r")

ax.scatter(1, 0, *color*="r")

ax.scatter(1, 1, *color*="g")

ax.set\_xlim([xmin, xmax])

ax.set\_ylim([-0.1, 1.1])

m = -p.weights[0] / p.weights[1]

c = -p.weights[2] / p.weights[1]

print(m, c)

ax.plot(X, m \* X + c )

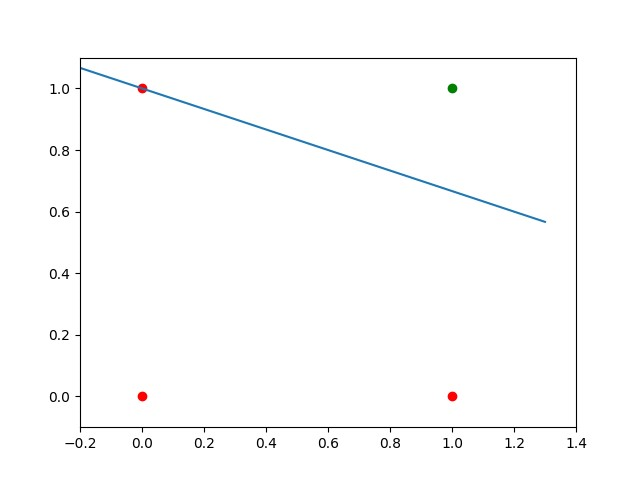
plt.plot()

plt.show()

Output:

Counter({'correct': 30})

-0.33333333333333326 1.0000000000000002



Name: Ayush Chanchal

Sap: 500097569

Batch: b6

EXPERIMENT: 09

Aim: Implementation of And, Or, Nand ,Nor and Not operations

**Describe:**

1. **AND Operation**:
   * The AND operation takes two binary inputs (0 or 1) and produces a 1 as output only if both inputs are 1. Otherwise, it produces a 0.
   * **Neural Network Interpretation**:
     + **Input Layer**: Two nodes representing the two binary inputs (x1, x2).
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias such that the output node activates (produces a 1) only when both inputs are 1.
2. **NAND Operation**:
   * The NAND operation is the opposite of AND. It produces a 0 as output only if both inputs are 1. Otherwise, it produces a 1.
   * **Neural Network Interpretation**:
     + **Input Layer**: Two nodes representing the two binary inputs (x1, x2).
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias such that the output node activates (produces a 1) only when both inputs are 0.
3. **OR Operation**:
   * The OR operation takes two binary inputs and produces a 1 as output if at least one of the inputs is 1.
   * **Neural Network Interpretation**:
     + **Input Layer**: Two nodes representing the two binary inputs (x1, x2).
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias such that the output node activates (produces a 1) when at least one input is 1.
4. **NOR Operation**:
   * The NOR operation is the opposite of OR. It produces a 0 as output if at least one of the inputs is 1.
   * **Neural Network Interpretation**:
     + **Input Layer**: Two nodes representing the two binary inputs (x1, x2).
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias such that the output node activates (produces a 1) when both inputs are 0.
5. **NOT Operation**:
   * The NOT operation takes a single binary input and produces the opposite output (1 becomes 0 and vice versa).
   * **Neural Network Interpretation**:
     + **Input Layer**: One node representing the binary input (x).
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias such that the output node activates (produces a 1) when the input is 0.
6. **XOR Operation**:
   * The XOR operation is exclusive OR. It produces a 1 as output if the inputs are different (one 1 and one 0). Otherwise, it produces a 0.
   * **Note**: XOR is not linearly separable and cannot be modelled with a single-layer perceptron. It requires a multi-layer perceptron.
   * **Neural Network Interpretation**:
     + **Input Layer**: Two nodes representing the two binary inputs (x1, x2).
     + **Hidden Layer**: One or more nodes.
     + **Output Layer**: One node producing the result.
   * **Weights and Bias**:
     + Assign appropriate weights and bias in both the hidden and output layers to model the XOR operation.

IMPLEMENTATION:

Source code:

import numpy as np

import matplotlib.pyplot as plt

def function(a):

if a>=0:

return 1

else:

return 0

def NeuralNetwork(x,w,b):

a=np.dot(x,w)+b

y=function(a)

return y

def Not\_Gate(x):

w=-1

b=0.5

return NeuralNetwork(x,w,b)

def And\_Gate(x):

w = np.array([1, 1])

b= -1.5

return NeuralNetwork(x, w, b)

def Or\_Gate(x):

w = np.array([1, 1])

b= -0.5

return NeuralNetwork(x, w, b)

def Nand\_Gate(x):

a=And\_Gate(x)

b=Not\_Gate(a)

return b

def Nor\_Gate(x):

a=Or\_Gate(x)

b=Not\_Gate(a)

return b

x1=0

x2=1

print("Not {}:".format(x1),Not\_Gate(x1))

print("Not {}:".format(x2),Not\_Gate(x2))

x1=np.array([0,0])

x2=np.array([0,1])

x3=np.array([1,0])

x4=np.array([1,1])

print("Or {}:".format(x1),Or\_Gate(x1))

print("Or {}:".format(x2),Or\_Gate(x2))

print("Or {}:".format(x3),Or\_Gate(x3))

print("Or {}:".format(x4),Or\_Gate(x4))

print("And {}:".format(x1),And\_Gate(x1))

print("And {}:".format(x2),And\_Gate(x2))

print("And {}:".format(x3),And\_Gate(x3))

print("And {}:".format(x4),And\_Gate(x4))

print("Nand {}:".format(x1),Nand\_Gate(x1))

print("Nand {}:".format(x2),Nand\_Gate(x2))

print("Nand {}:".format(x3),Nand\_Gate(x3))

print("Nand {}:".format(x4),Nand\_Gate(x4))

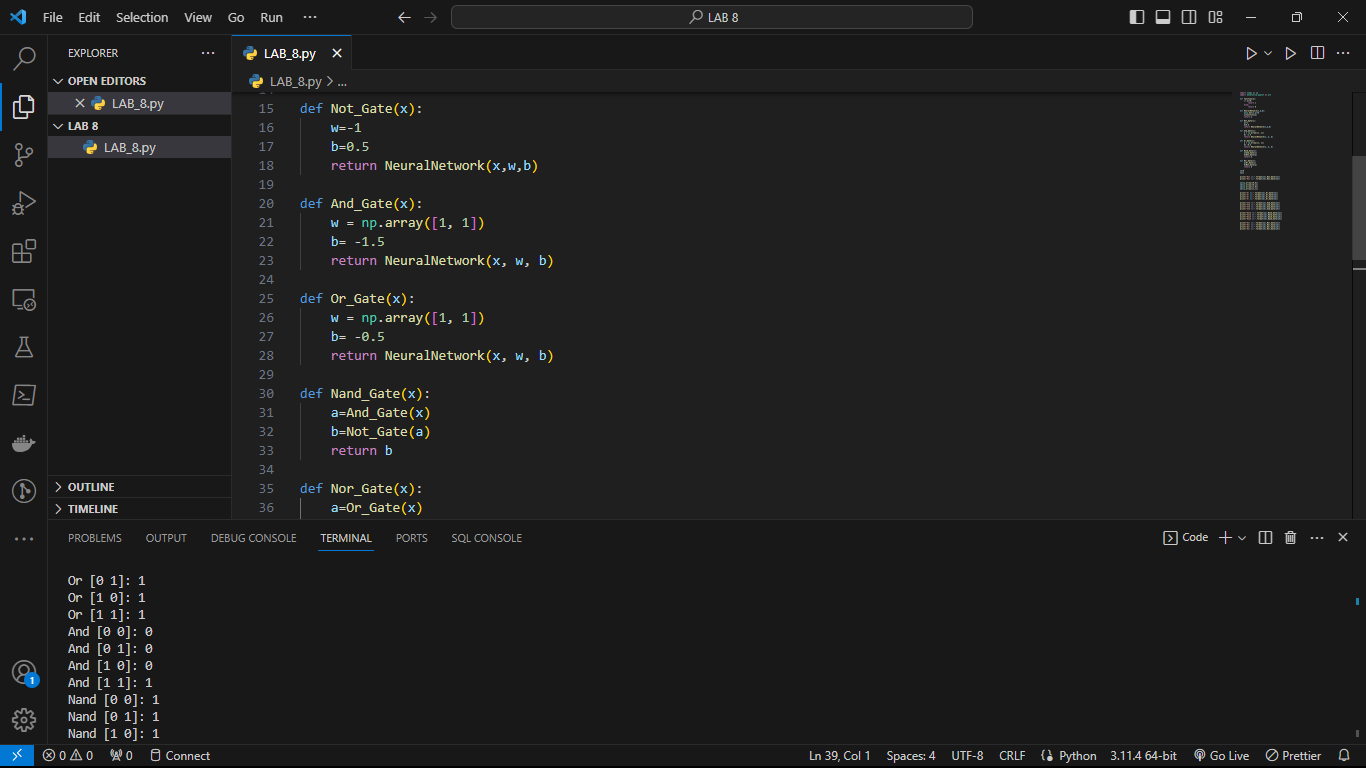
print("Nor {}:".format(x1),Nor\_Gate(x1))

print("Nor {}:".format(x2),Nor\_Gate(x2))

print("Nor {}:".format(x3),Nor\_Gate(x3))

print("Nor {}:".format(x4),Nor\_Gate(x4))

Screenshot:



NAME: AYUSH CHANCHAL

SAP ID:500097569

BATCH: B6

EXPERIMENT:10

Aim: LSTM Implementation in python

Q). Explain LSTM?

: - Traditional RNNs may struggle to capture long-range dependencies in sequences because they suffer from the vanishing gradient problem. This means that as information passes through many time steps, it can quickly become small and eventually disappear, making it difficult for the network to learn from distant events. LSTM solves this issue by introducing a more complex structure compared to simple RNN. They have memory cells, which are like conveyor belts that can store information and move it over several time steps. This allows LSTMs to selectively remember or forget information over time.

Key Components of LSTM:

1. Cell state
2. Hidden State
3. Gates
4. Gate Activation function

Source code:

import numpy as np

import pandas as  pd

import matplotlib.pyplot as plt

from keras.preprocessing.sequence import TimeseriesGenerator

from sklearn.preprocessing import MinMaxScaler

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LSTM

from sklearn.metrics import mean\_squared\_error

from math import sqrt

import io

df = pd.read\_csv(io.BytesIO(uploaded['airline-passengers.csv']),index\_col='Month',parse\_dates=True)

# df=pd.read\_csv('airline-passengers.csv',index\_col='Month',parse\_dates=True)

df.index.freq='MS'

print(df.shape)

print(df.columns)

plt.figure(figsize=(20,40))

plt.plot(df.Passengers,linewidth=2)

plt.show()

nobs=12

df\_train=df.iloc[:-nobs]

df\_test=df.iloc[-nobs:]

print(df\_train)

print(df\_test)

scaler=MinMaxScaler()

scaler.fit(df\_train)

scaled\_train=scaler.transform(df\_train)

scaled\_test=scaler.transform(df\_test)

n\_inputs=12

n\_features=1

generator=TimeseriesGenerator(scaled\_train,scaled\_train,length=n\_inputs,batch\_size=1)

for i in range(len(generator)):

    x,y=generator[i]

    print(f'\n{x.flatten()} and {y}')

print(x.shape)

model=Sequential()

model.add(LSTM(200,activation='relu',input\_shape=(n\_inputs,n\_features)))

model.add(Dense(1))

model.compile(optimizer='adam',loss='mse')

print(model.summary())

model.fit(generator,epochs=50)

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

last\_train\_batch=scaled\_train[-12:]

last\_train\_batch=last\_train\_batch.reshape(1,12,1)

print(last\_train\_batch)

model.predict(last\_train\_batch)

scaled\_test[0]

y\_pred=[]

first\_batch=scaled\_train[-n\_inputs:]

current\_batch=first\_batch.reshape(1,n\_inputs,n\_features)

for i in range(len(scaled\_test)):

    batch=current\_batch

    pred=model.predict(batch)[0]

    y\_pred.append(pred)

    current\_batch=np.append(current\_batch[:,1:, :],[[pred]],axis=1)

print(y\_pred)

print(scaled\_test)

print(df\_test)

y\_pred\_transformed=scaler.inverse\_transform(y\_pred)

y\_pred\_transformed=np.round(y\_pred\_transformed,0)

y\_pred\_final=y\_pred\_transformed.astype(int)

print(y\_pred\_final)

print(df\_test.values)

print(y\_pred\_final)

df\_test['Predictions']=y\_pred\_final

print(df\_test)

plt.figure(figsize=(15,6))

plt.plot(df\_train.index,df\_train.Passengers,linewidth=2,color='black',label='Train Values')

plt.plot(df\_test.index,df\_test.Passengers,linewidth=2,color='green',label='True Values')

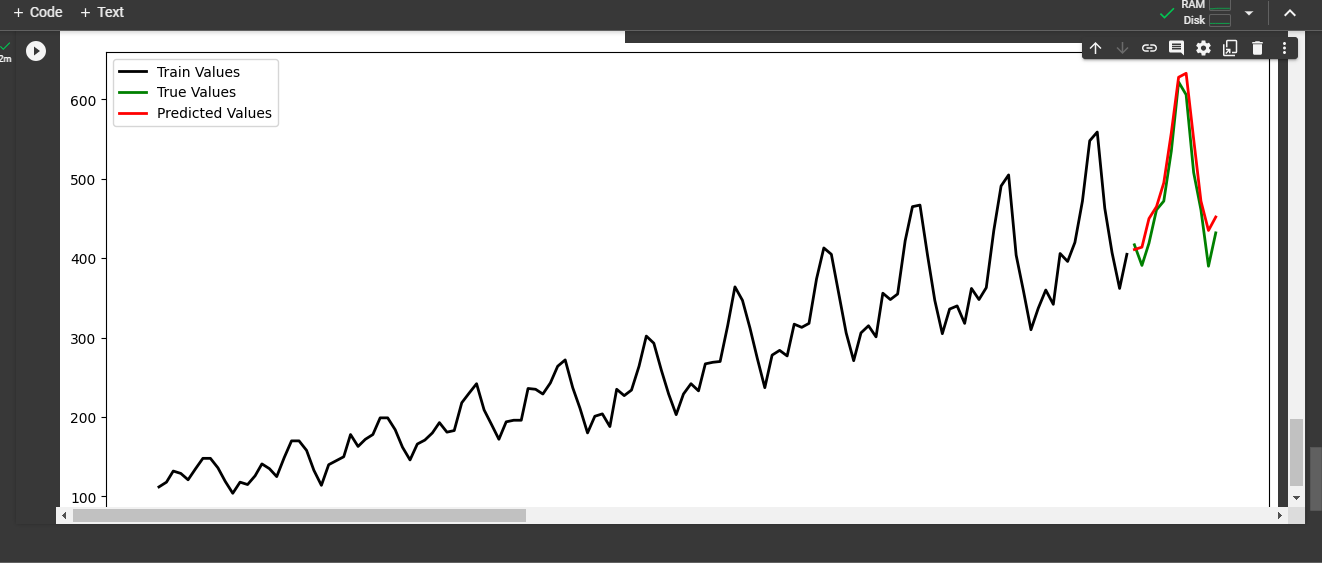
plt.plot(df\_test.index,df\_test.Predictions,linewidth=2,color='red',label='Predicted Values')

plt.legend()

plt.show()

squareroot=sqrt(mean\_squared\_error(df\_test.Passengers,df\_test.Predictions))

Screenshot:



*Name: Ayush Chanchal*

*Sap id: 500097569*

*Batch: B6*

*EXPERIMENT: 11*

Aim: RBF implementation

Q. Explain RBF (Radial Basis Function)

: = A radial basis function (RBF) neural network is a type of artificial neural network that uses radial basis functions as activation functions. It typically consists of three layers: an input layer, a hidden layer, and an output layer. The hidden layer applies a radial basis function, usually a Gaussian function, to the input. The output layer then linearly combines these outputs to generate the final output. RBF neural networks are highly versatile and are extensively used in pattern classification tasks, function approximation, and a variety of machine learning applications. They are especially known for their ability to handle non-linear problems effectively.

**Structure of RBF neural networks**

An RBF neural network typically comprises three layers:

• Input layer: This layer simply transmits the inputs to the neurons in the hidden layer.

• Hidden layer: Each neuron in this layer applies a radial basis function to the inputs it receives.

• Output layer: Each neuron in this layer computes a weighted sum of the outputs from the hidden layer, resulting in the final output.

Source Code:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

from sklearn.datasets import make\_classification

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import accuracy\_score

import scipy.spatial.distance as distance

class RadialBasisFunctionNeuralNetwork:

def \_\_init\_\_(self, num\_of\_rbf\_units=10):

self.num\_of\_rbf\_units = num\_of\_rbf\_units

def \_rbf\_unit(self, rbf\_center, point\_in\_dataset):

return np.exp(-self.beta \* distance.cdist([point\_in\_dataset], [rbf\_center],'euclidean')\*\*2).flatten()[0]

def \_construct\_interpolation\_matrix(self, input\_dataset):

interpolation\_matrix = np.zeros((len(input\_dataset), self.num\_of\_rbf\_units))

for idx, point\_in\_dataset in enumerate(input\_dataset):

for center\_idx, rbf\_center in enumerate(self.rbf\_centers):

interpolation\_matrix[idx, center\_idx] = self.\_rbf\_unit(rbf\_center, point\_in\_dataset)

return interpolation\_matrix

def train\_model(self, input\_dataset, target\_dataset):

self.kmeans\_clustering = KMeans(n\_clusters=self.num\_of\_rbf\_units,

random\_state=0).fit(input\_dataset)

self.rbf\_centers = self.kmeans\_clustering.cluster\_centers\_

self.beta = 1.0 / (2.0 \* (self.kmeans\_clustering.inertia\_ / input\_dataset.shape[0]))

interpolation\_matrix = self.\_construct\_interpolation\_matrix(input\_dataset)

self.model\_weights =np.linalg.pinv(interpolation\_matrix.T.dot(interpolation\_matrix)).dot(interpolation\_matrix.T).dot(target\_dataset)

def predict(self, input\_dataset):

interpolation\_matrix = self.\_construct\_interpolation\_matrix(input\_dataset)

predicted\_values = interpolation\_matrix.dot(self.model\_weights)

return predicted\_values

if \_\_name\_\_ == "\_\_main\_\_":

# Generating a simple classification dataset

input\_dataset, target\_dataset = make\_classification(n\_samples=500, n\_features=2,n\_informative=2, n\_redundant=0, n\_classes=2)

# Initializing and training the RBF neural network

rbf\_neural\_network = RadialBasisFunctionNeuralNetwork(num\_of\_rbf\_units=20)

rbf\_neural\_network.train\_model(input\_dataset, target\_dataset)

# Predicting the target values

predictions = rbf\_neural\_network.predict(input\_dataset)

# Converting continuous output to binary labels

binary\_predictions = np.where(predictions > 0.5, 1, 0)

# print("Accuracy: {}".format(accuracy\_score(target\_dataset, binary\_predictions)))

print(f"Accuracy: {accuracy\_score(target\_dataset, binary\_predictions)\*100}%")

# Plotting the results

plt.scatter(input\_dataset[:, 0], input\_dataset[:, 1], c=binary\_predictions, cmap='viridis',alpha=0.7)

plt.scatter(rbf\_neural\_network.rbf\_centers[:, 0], rbf\_neural\_network.rbf\_centers[:, 1], c='red')

plt.title('Classification Result')

plt.show()

Screenshot:

