**Practical 1**

**Aim:** Design a simple linear neural network model Calculate the output of neural net using both binary and bipolar sigmoidal function.

**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: "))

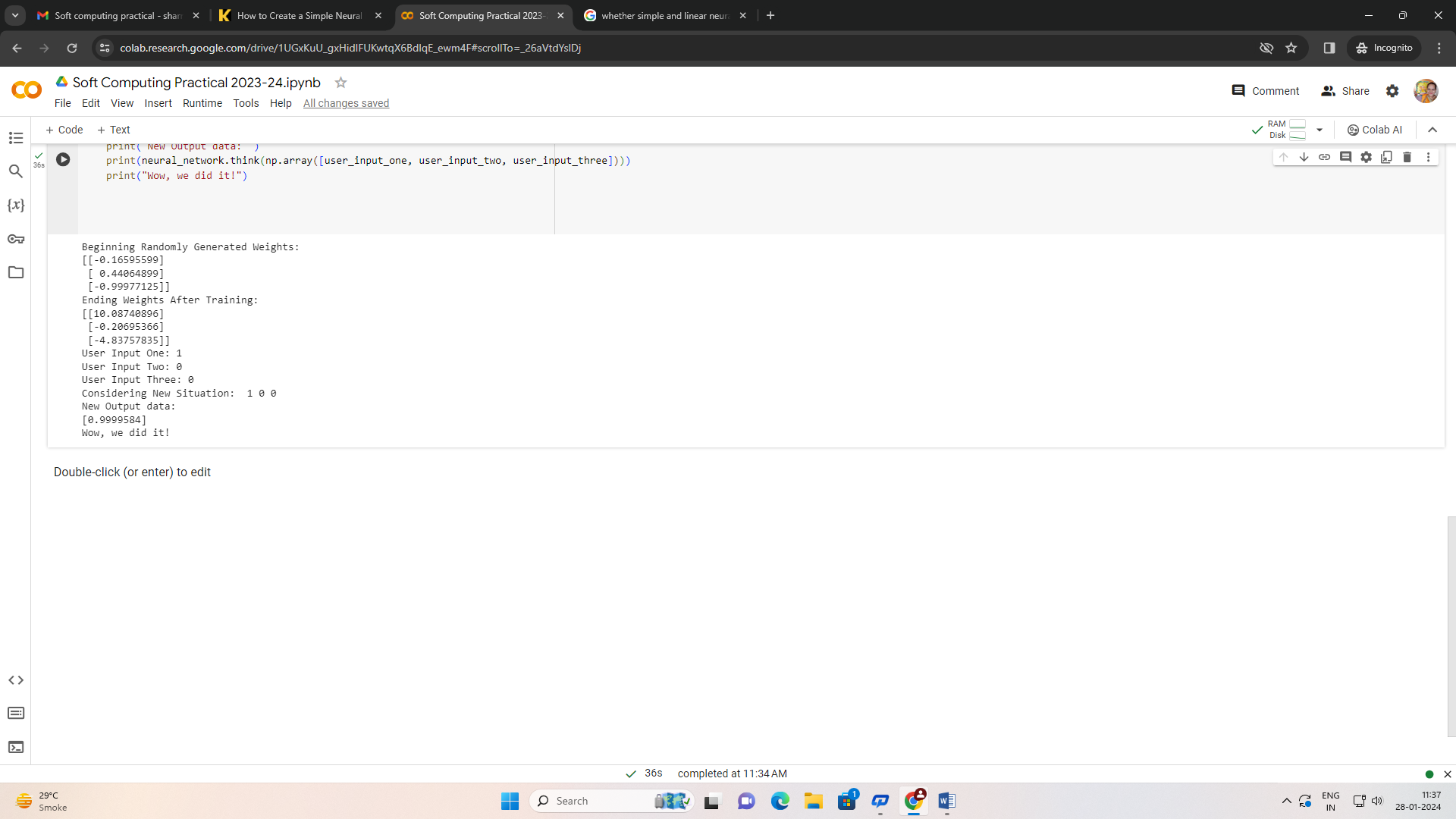
print("Considering New Situation: ", user\_input\_one, user\_input\_two, user\_input\_three)

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:**



**Conclusion:** Thus implemented the experiment successfully

**Practical 2.**

**Aim:-** Generate XOR function using McCulloch-Pitts neural net

**Code:**

# import Python Libraries

import numpy as np

from matplotlib import pyplot as plt

# Sigmoid Function

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

# Initialization of the neural network parameters

# Initialized all the weights in the range of between 0 and 1

# Bias values are initialized to 0

def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):

    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)

    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)

    b1 = np.zeros((neuronsInHiddenLayers, 1))

    b2 = np.zeros((outputFeatures, 1))

    parameters = {"W1" : W1, "b1": b1,

                  "W2" : W2, "b2": b2}

    return parameters

# Forward Propagation

def forwardPropagation(X, Y, parameters):

    m = X.shape[1]

    W1 = parameters["W1"]

    W2 = parameters["W2"]

    b1 = parameters["b1"]

    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1

    A1 = sigmoid(Z1)

    Z2 = np.dot(W2, A1) + b2

    A2 = sigmoid(Z2)

    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)

    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))

    cost = -np.sum(logprobs) / m

    return cost, cache, A2

# Backward Propagation

def backwardPropagation(X, Y, cache):

    m = X.shape[1]

    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache

    dZ2 = A2 - Y

    dW2 = np.dot(dZ2, A1.T) / m

    db2 = np.sum(dZ2, axis = 1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)

    dZ1 = np.multiply(dA1, A1 \* (1- A1))

    dW1 = np.dot(dZ1, X.T) / m

    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,

                 "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients

# Updating the weights based on the negative gradients

def updateParameters(parameters, gradients, learningRate):

    parameters["W1"] = parameters["W1"] - learningRate \* gradients["dW1"]

    parameters["W2"] = parameters["W2"] - learningRate \* gradients["dW2"]

    parameters["b1"] = parameters["b1"] - learningRate \* gradients["db1"]

    parameters["b2"] = parameters["b2"] - learningRate \* gradients["db2"]

    return parameters

# Model to learn the XOR truth table

X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XOR input

Y = np.array([[0, 1, 1, 0]]) # XOR output

# Define model parameters

neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)

inputFeatures = X.shape[0] # number of input features (2)

outputFeatures = Y.shape[0] # number of output features (1)

parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures)

epoch = 100000

learningRate = 0.01

losses = np.zeros((epoch, 1))

for i in range(epoch):

    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)

    gradients = backwardPropagation(X, Y, cache)

    parameters = updateParameters(parameters, gradients, learningRate)

# Evaluating the performance

plt.figure()

plt.plot(losses)

plt.xlabel("EPOCHS")

plt.ylabel("Loss value")

plt.show()

# Testing

X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input

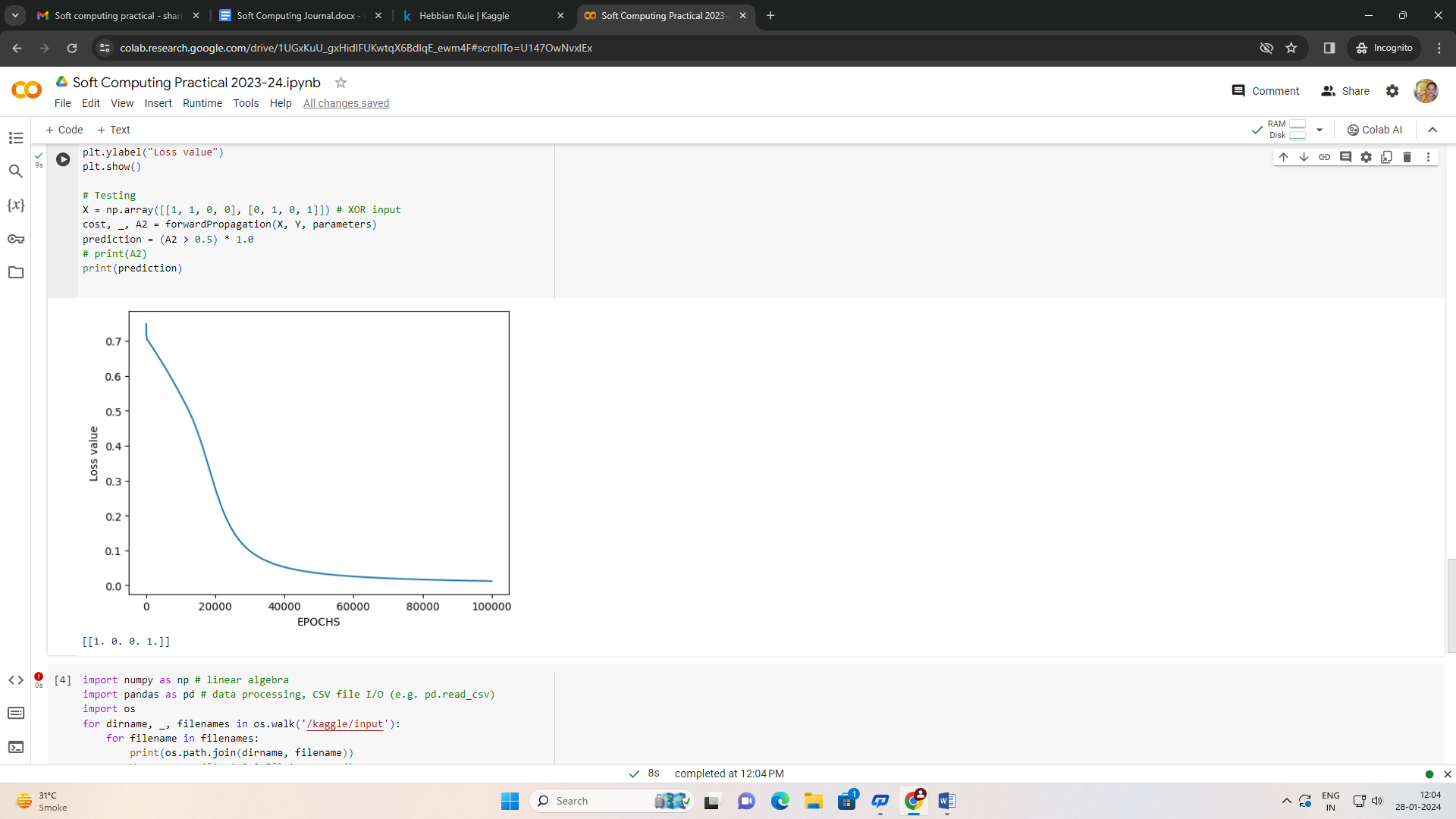
cost, \_, A2 = forwardPropagation(X, Y, parameters)

prediction = (A2 > 0.5) \* 1.0

# print(A2)

print(prediction)

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 3.**

**Aim:-** Write a program to implement Hebb’s rule.

**Code:**

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)

import os

for dirname, \_, filenames in os.walk('/kaggle/input'):

    for filename in filenames:

        print(os.path.join(dirname, filename))

W = np.array([1,-1,0,0.5]).transpose()

Xi = [np.array([1,-2,1.5,0]).transpose(),np.array([1,-0.5,-2,-1.5]).transpose(), np.array([0,1,-1,1.5]).transpose()]

c = 1  #Learning constant

Iteration = 0

for i in range(len(Xi)):

    net = sum(W.transpose()\*Xi[i])

    Fnet = np.sign(net)

    dw = c \* Fnet \* Xi[i]

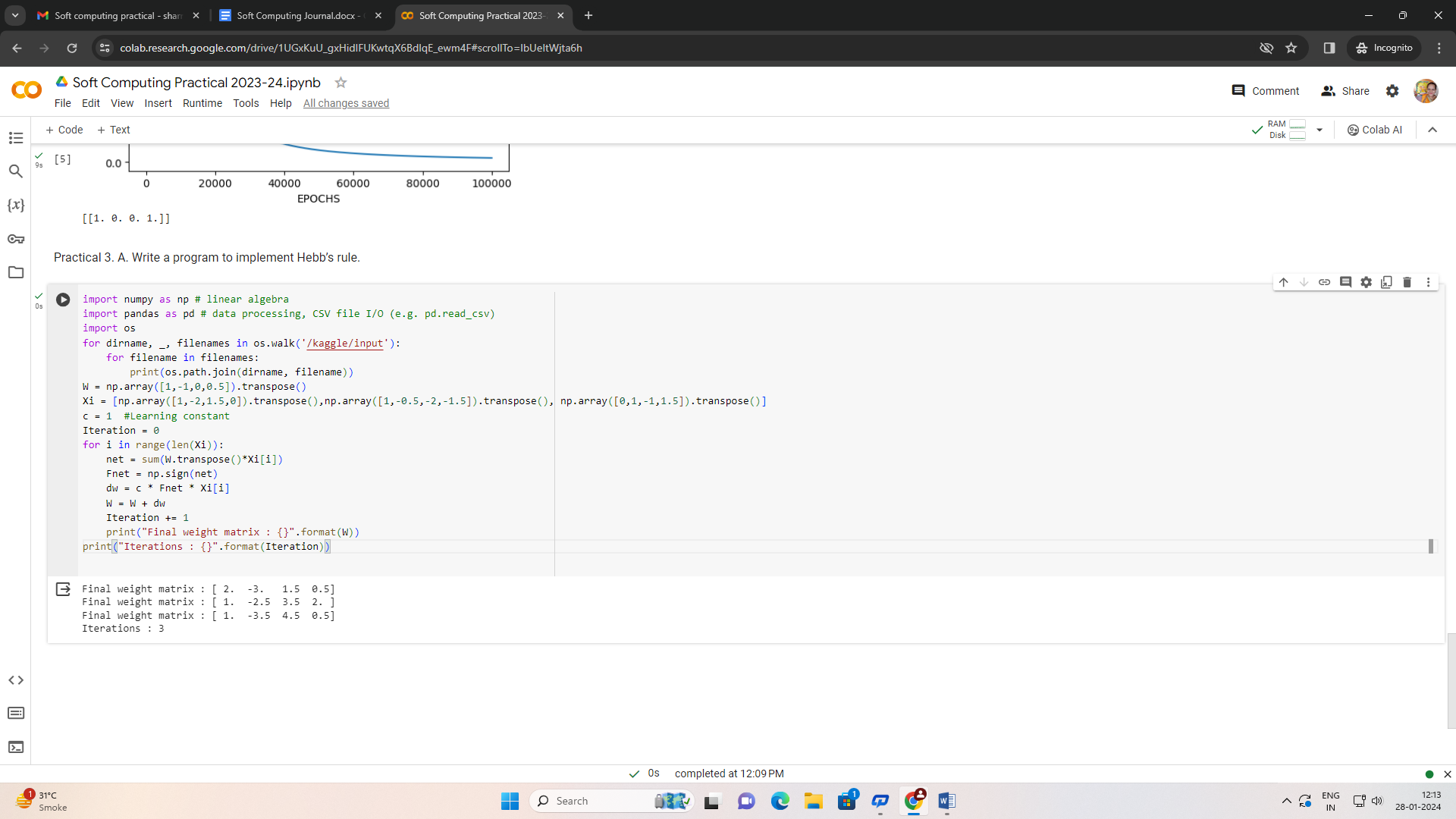
    W = W + dw

    Iteration += 1

    print("Final weight matrix : {}".format(W))

print("Iterations : {}".format(Iteration))

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 4.**

**Aim:-** Write a program for Back Propagation Algorithm

**Code:**

import pandas as pd

url = "https://raw.githubusercontent.com/jigsawlabs-student/pytorch-intro-curriculum/main/5-training-mathematically/cell\_multiple.csv"

df = pd.read\_csv(url)

df[:2]

import torch

X\_tensor = torch.tensor(df[['mean\_area', 'mean\_concavity']].values).float()

y\_tensor = torch.tensor(df['is\_cancerous']).float()

first\_x = X\_tensor[0]

first\_y = y\_tensor[0]

first\_x, first\_y

def linear\_fn(x, w, b):

    return x @ w + b

def activation\_fn(z):

    return torch.sigmoid(z)

w = torch.tensor([.5, .3]).float()

b = torch.tensor(-2.).float()

z = linear\_fn(first\_x, w, b)

z

y\_hat = activation\_fn(z)

y\_hat

import torch

def delta\_J\_delta\_sigma(y\_hat, y):

    return torch.sum(2\*(y\_hat - y))

def delta\_sigma\_delta(z):

    return torch.sigmoid(z)\*(1 - torch.sigmoid(z))

def delta\_z\_delta\_w(x):

    return x

def delta\_z\_delta\_b():

    return 1

w\_grad = delta\_z\_delta\_w(first\_x)\*delta\_sigma\_delta(z)\*delta\_J\_delta\_sigma(y\_hat, first\_y)

w\_grad

b\_grad = delta\_z\_delta\_b()\*delta\_sigma\_delta(z)\*delta\_J\_delta\_sigma(y\_hat, first\_y)

b\_grad

dj\_dsig = delta\_J\_delta\_sigma(y\_hat, first\_y)

dj\_dsig

dsig\_dz = delta\_sigma\_delta(z)

dz\_dJ = dsig\_dz\*dj\_dsig

dz\_dJ

dz\_dw = delta\_z\_delta\_w(first\_x)

dw\_dJ = dz\_dw\*dz\_dJ

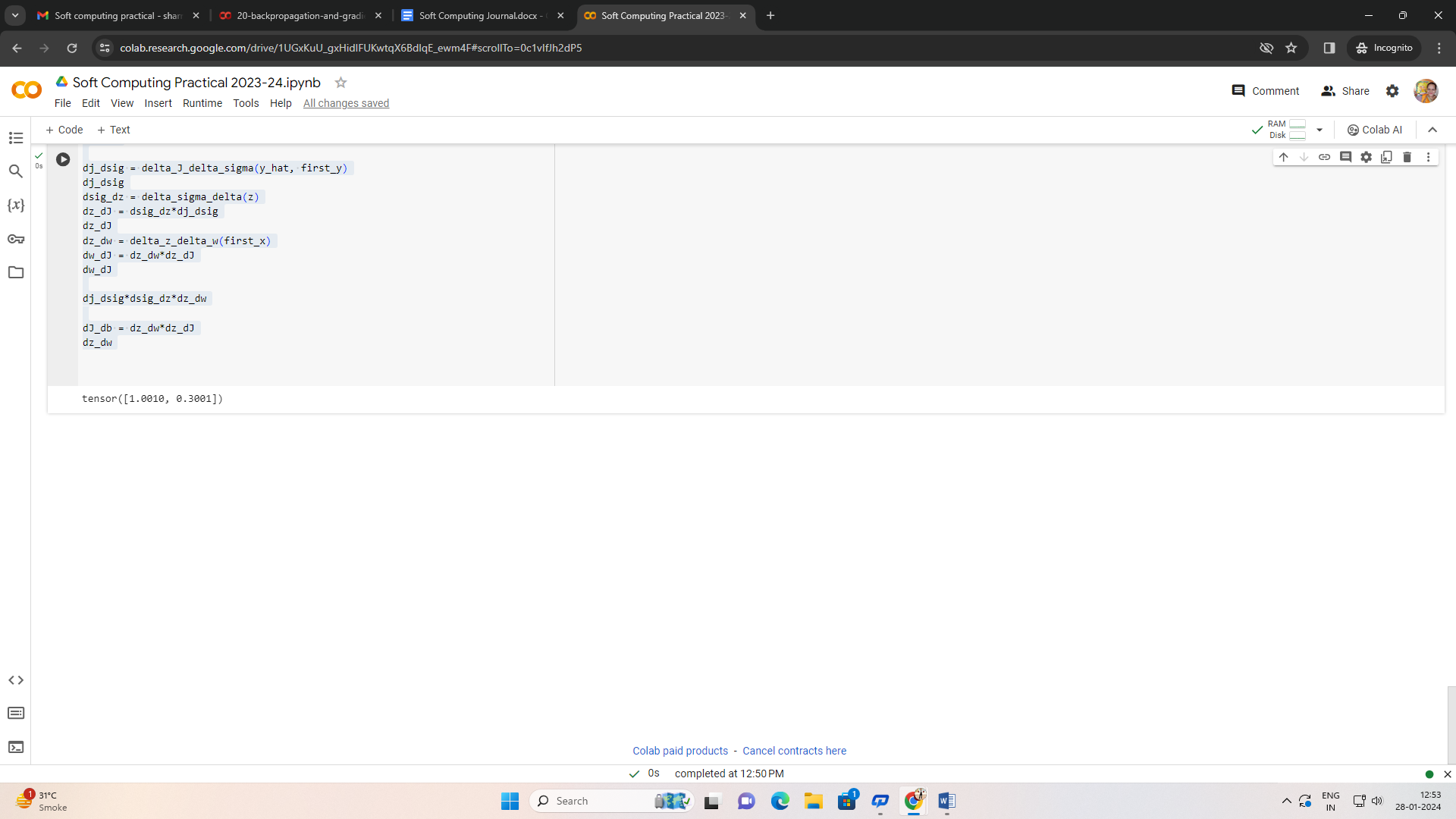
dw\_dJ

dj\_dsig\*dsig\_dz\*dz\_dw

dJ\_db = dz\_dw\*dz\_dJ

dz\_dw

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 5.**

**Aim:-** Write a program for Hopfield Network

**Code:**

import numpy as np

import matplotlib.pyplot as plt

import torch

import torchvision

from torchvision import transforms

from copy import deepcopy

import torch.nn.functional as F

def load\_mnist(batch\_size,norm\_factor=1):

    transform = transforms.Compose([transforms.ToTensor()])

    trainset = torchvision.datasets.MNIST(root='./mnist\_data', train=True,

                                            download=True, transform=transform)

    print("trainset: ", trainset)

    trainloader = torch.utils.data.DataLoader(trainset, batch\_size=batch\_size,

                                            shuffle=True)

    print("trainloader: ", trainloader)

    trainset = list(iter(trainloader))

    testset = torchvision.datasets.MNIST(root='./mnist\_data', train=False,

                                        download=True, transform=transform)

    testloader = torch.utils.data.DataLoader(testset, batch\_size=batch\_size,

                                            shuffle=True)

    testset = list(iter(testloader))

    for i,(img, label) in enumerate(trainset):

        trainset[i] = (img.reshape(len(img),784) /norm\_factor ,label)

    for i,(img, label) in enumerate(testset):

        testset[i] = (img.reshape(len(img),784) /norm\_factor ,label)

    return trainset, testset

trainset, testset = load\_mnist(1000)

def binarize(img):

  i = deepcopy(img)

  i[img > 0] = -1

  i[img <=0] = 1

  return i

def zero\_bottom\_half(img):

  i = deepcopy(img)

  H,W = img.shape

  i[H//2:H,:] = -1

  return i

imgs,labels = trainset[0]

digit = imgs[3,:].reshape(784,1)

digit = binarize(digit)

halved\_digit = zero\_bottom\_half(digit.reshape(28,28)).reshape(784,1)

plt.subplot(1,2,1)

plt.title("Full image")

plt.imshow(digit.reshape(28,28))

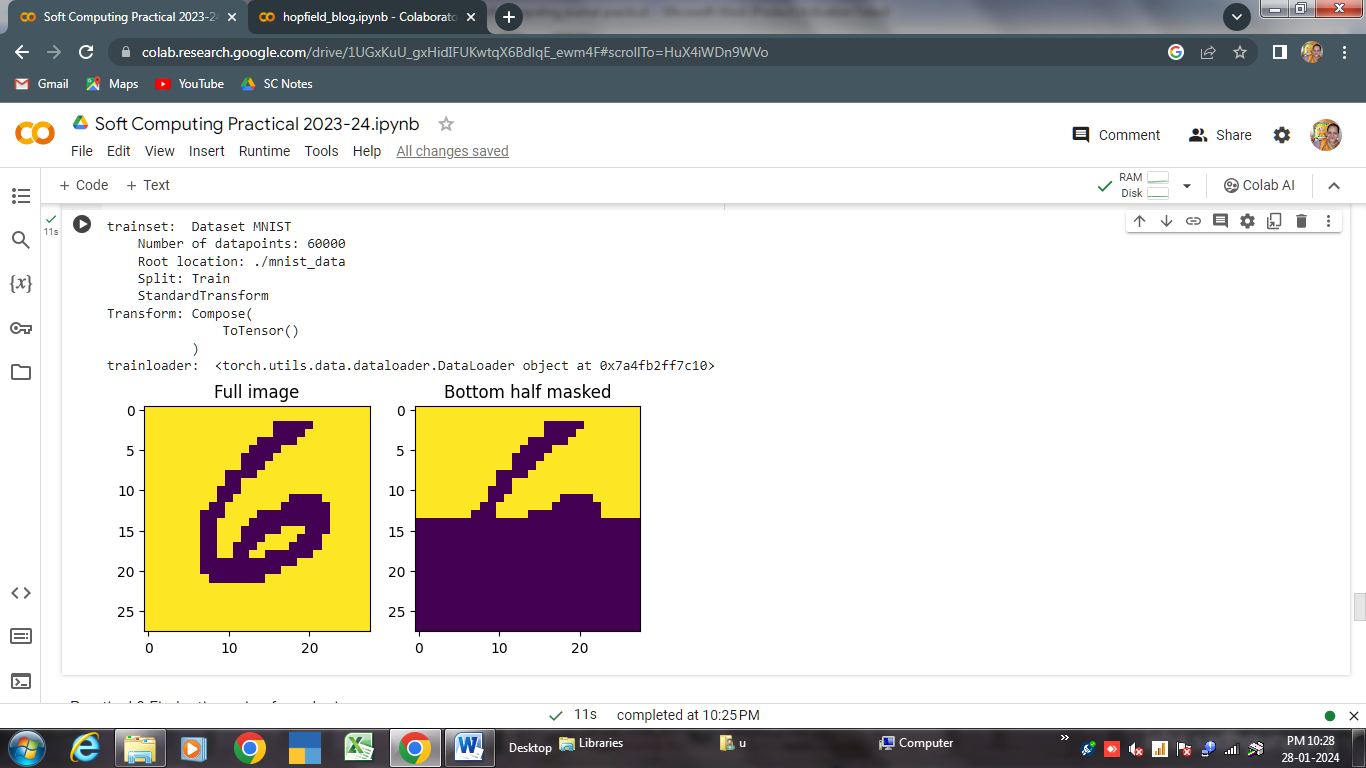
plt.subplot(1,2,2)

plt.title("Bottom half masked")

plt.imshow(halved\_digit.reshape(28,28))

plt.show()

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 6.**

**Aim:-** Kohonen Self organizing map

**Code:**

import math

class SOM:

    # Function here computes the winning vector

    # by Euclidean distance

    def winner(self, weights, sample):

        D0 = 0

        D1 = 0

        for i in range(len(sample)):

            D0 = D0 + math.pow((sample[i] - weights[0][i]), 2)

            D1 = D1 + math.pow((sample[i] - weights[1][i]), 2)

        # Selecting the cluster with smallest distance as winning cluster

        if D0 < D1:

            return 0

        else:

            return 1

    # Function here updates the winning vector

    def update(self, weights, sample, J, alpha):

        # Here iterating over the weights of winning cluster and modifying them

        for i in range(len(weights[0])):

            weights[J][i] = weights[J][i] + alpha \* (sample[i] - weights[J][i])

        return weights

# Driver code

def main():

    # Training Examples ( m, n )

    T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]

    m, n = len(T), len(T[0])

    # weight initialization ( n, C )

    weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]

    # training

    ob = SOM()

    epochs = 3

    alpha = 0.5

    for i in range(epochs):

        for j in range(m):

            # training sample

            sample = T[j]

            # Compute winner vector

            J = ob.winner(weights, sample)

            # Update winning vector

            weights = ob.update(weights, sample, J, alpha)

    # classify test sample

    s = [0, 0, 0, 1]

    J = ob.winner(weights, s)

    print("Test Sample s belongs to Cluster : ", J)

    print("Trained weights : ", weights)

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

    main()

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 7.**

**Aim:-** Write a program for Linear separation

**Code:**

from sklearn import svm

import numpy as np

# Making dataset

X = np.array([[1, 2], [2, 3], [3, 1], [4, 3]])

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

# Now lets train svm model

model = svm.SVC(kernel='linear')

model.fit(X, Y)

# Lets predict for new input

n\_data = np.array([[5, 2], [2, 1]])

pred = model.predict(n\_data)

print(pred)



**We will plot boundary for the same**

import matplotlib.pyplot as plt

# lets plot decision boundary for this

w = model.coef\_[0]

b = model.intercept\_[0]

x = np.linspace(1, 4)

y = -(w[0] / w[1]) \* x - b / w[1]

plt.plot(x, y, 'k-')

# plot data points

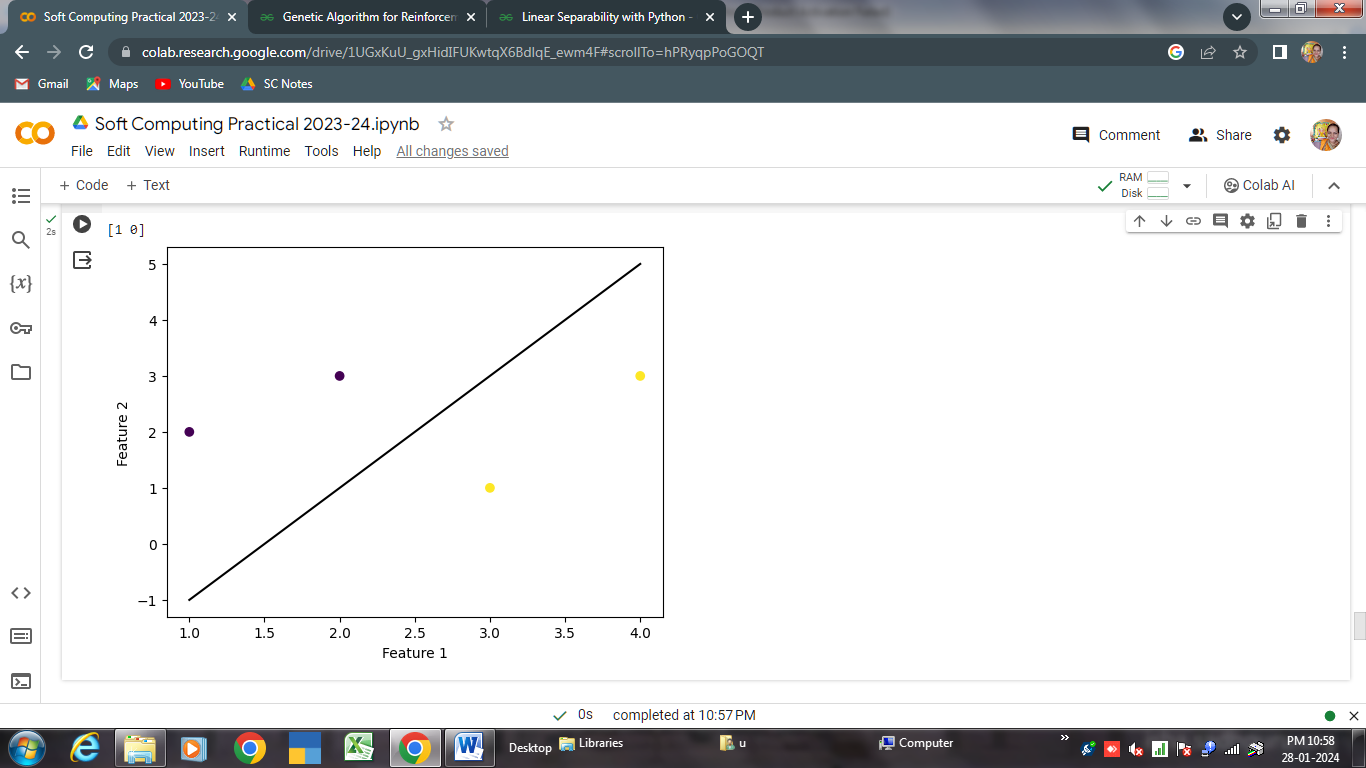
plt.scatter(X[:, 0], X[:, 1], c=Y)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 9.**

**Aim:-** Find ratios using fuzzy logic

**Code:**

!pip install fuzzywuzzy

!pip install python-Levenshtein

from fuzzywuzzy import fuzz

from fuzzywuzzy import process

s1 = "I love fuzzysforfuzzys"

s2 = "I am loving fuzzysforfuzzys"

print ("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))

print ("FuzzyWuzzy PartialRatio: ", fuzz.partial\_ratio(s1, s2))

print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token\_sort\_ratio(s1, s2))

print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token\_set\_ratio(s1, s2))

print ("FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n')

# for process library,

query = 'fuzzys for fuzzys'

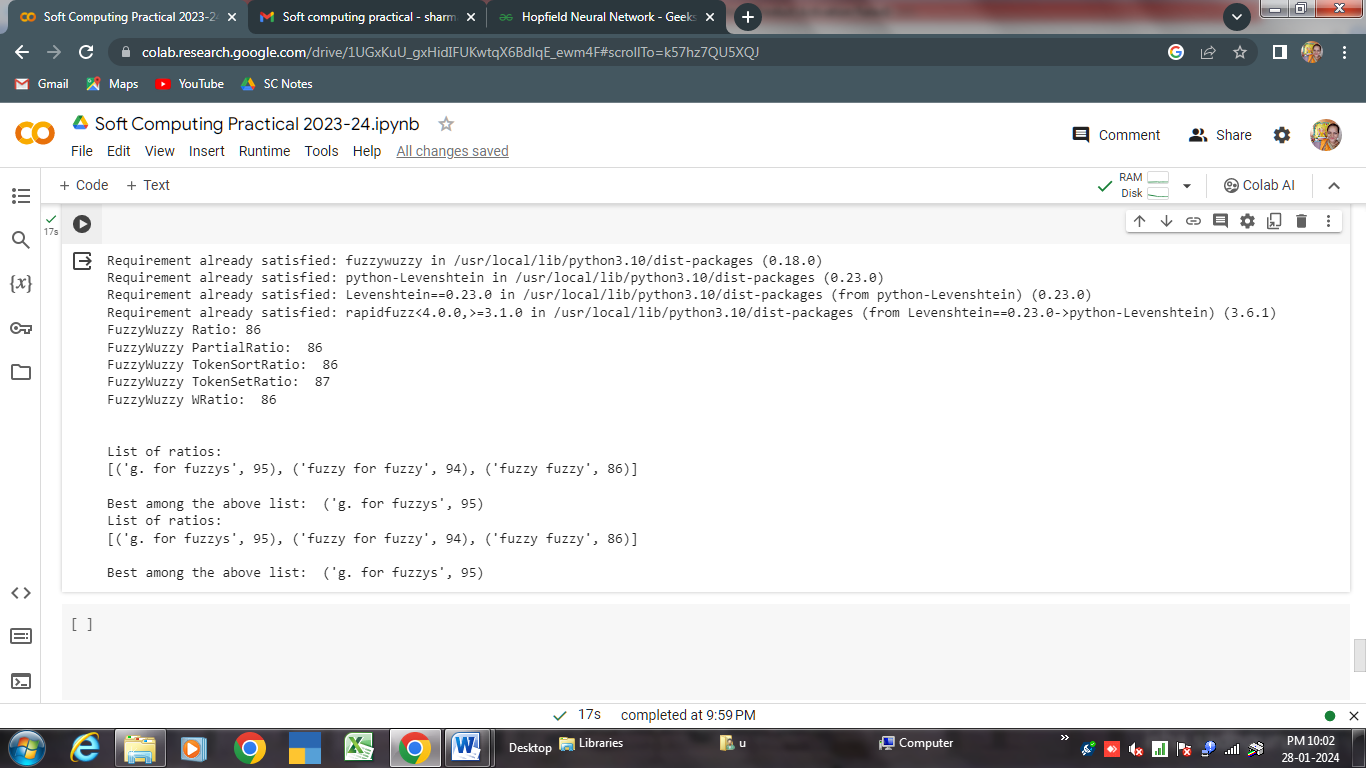
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']

print ("List of ratios: ")

print (process.extract(query, choices), '\n')

print ("Best among the above list: ",process.extractOne(query, choices))

**Output:**



**Conclusion**: Thus implemented the experiment successfully

**Practical 10.**

**Aim:-** Implementation of Simple Genetic Algorithm

**Code:**

import numpy as np

import matplotlib.pyplot as plt

# specifying the size for each and

# every matplotlib plot globally

plt.rcParams['figure.figsize'] = [8, 6]

# defining list objects with range of the graph

x1\_range = [-100, 100]

x2\_range = [-100, 100]

# empty list object to store the population

population = []

# this function is used to generate the population

# and appending it to the population list defined above

# it takes the attributes as no. of features in a

# population and size that we have in it

def populate(features, size = 1000):

  # here we are defining the coordinate

  # for each entity in a population

  initial = []

  for \_ in range(size):

    entity = []

    for feature in features:

      # this \* feature variable unpacks a list

      # or tuple into position arguments.

      val = np.random.randint(\*feature)

      entity.append(val)

    initial.append(entity)

  return np.array(initial)

# defining the virus in the form of numpy array

virus = np.array([5, 5])

# only the 100 fit ones will survive in this one

def fitness(population, virus, size = 100):

  scores = []

  # enumerate also provides the index as for the iterator

  for index, entity in enumerate(population):

    score = np.sum((entity-virus)\*\*2)

    scores.append((score, index))

  scores = sorted(scores)[:size]

  return np.array(scores)[:, 1]

# this function is used to plot the graph

def draw(population, virus):

  plt.xlim((-100, 100))

  plt.ylim((-100, 100))

  plt.scatter(population[:, 0], population[:, 1], c ='green', s = 12)

  plt.scatter(virus[0], virus[1], c ='red', s = 60)

def reduction(population, virus, size = 100):

  # only the index of the fittest ones

  # is returned in sorted format

  fittest = fitness(population, virus, size)

  new\_pop = []

  for item in fittest:

    new\_pop.append(population[item])

  return np.array(new\_pop)

# cross mutation in order to generate the next generation

# of the population which will be more immune to virus than previous

def cross(population, size = 1000):

  new\_pop = []

  for \_ in range(size):

    p = population[np.random.randint(0, len(population))]

    m = population[np.random.randint(0, len(population))]

    # we are only considering half of each

    # without considering random selection

    entity = []

    entity.append(\*p[:len(p)//2])

    entity.append(\*m[len(m)//2:])

    new\_pop.append(entity)

  return np.array(new\_pop)

# generating and adding the random features to

# the entity so that it looks more distributed

def mutate(population):

  return population + np.random.randint(-10, 10, 2000).reshape(1000, 2)

# the complete cycle of the above steps

population = populate([x1\_range, x2\_range], 1000)

# gens is the number of generation

def cycle(population, virus, gens = 1):

  # if we change the value of gens, we'll get

  # next and genetically more powerful generation

  # of the population

  for \_ in range(gens):

    population = reduction(population, virus, 100)

    population = cross(population, 1000)

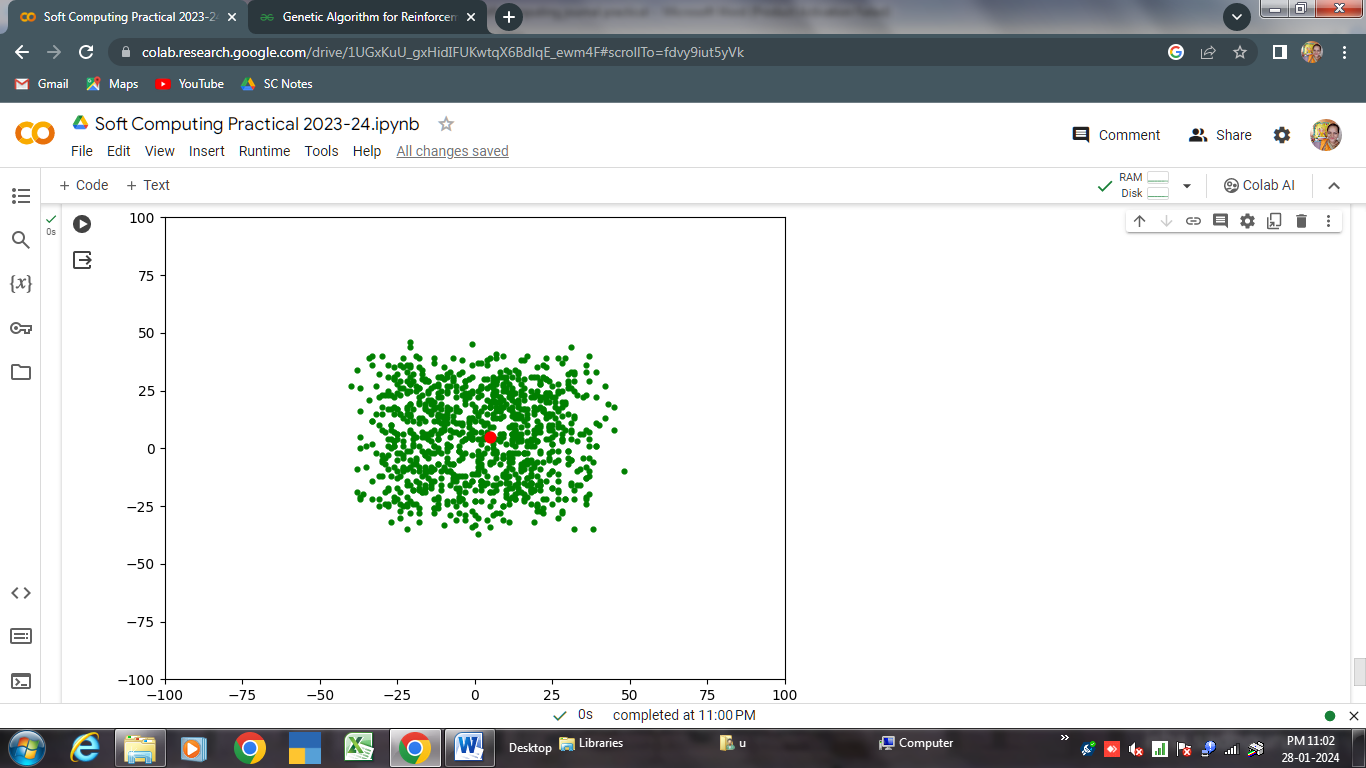
    population = mutate(population)

  return population

population = cycle(population, virus)

draw(population, virus)

**Output:**



**Conclusion**: Thus implemented the experiment successfully