# Practical No: 01

**Aim: A)** **Design a simple linear neural network model.**

#include<iostream.h>

#include<conio.h>

void main()

{

clrscr();

float x,b,w,net;

float out;

cout<<"\nEnter the input X=";

cin>>x;

cout<<"\nEnter the bias b=";

cin>>b;

cout<<"\nEnter the weight W=";

cin>>w;

net=(w\*x+b);

cout<<"\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OutPut\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout<<"\nnet="<<net<<endl;

if(net<0)

{

out=0;

}

else

if((net>=0)&&(net<=1))

{

out=net;

}

else

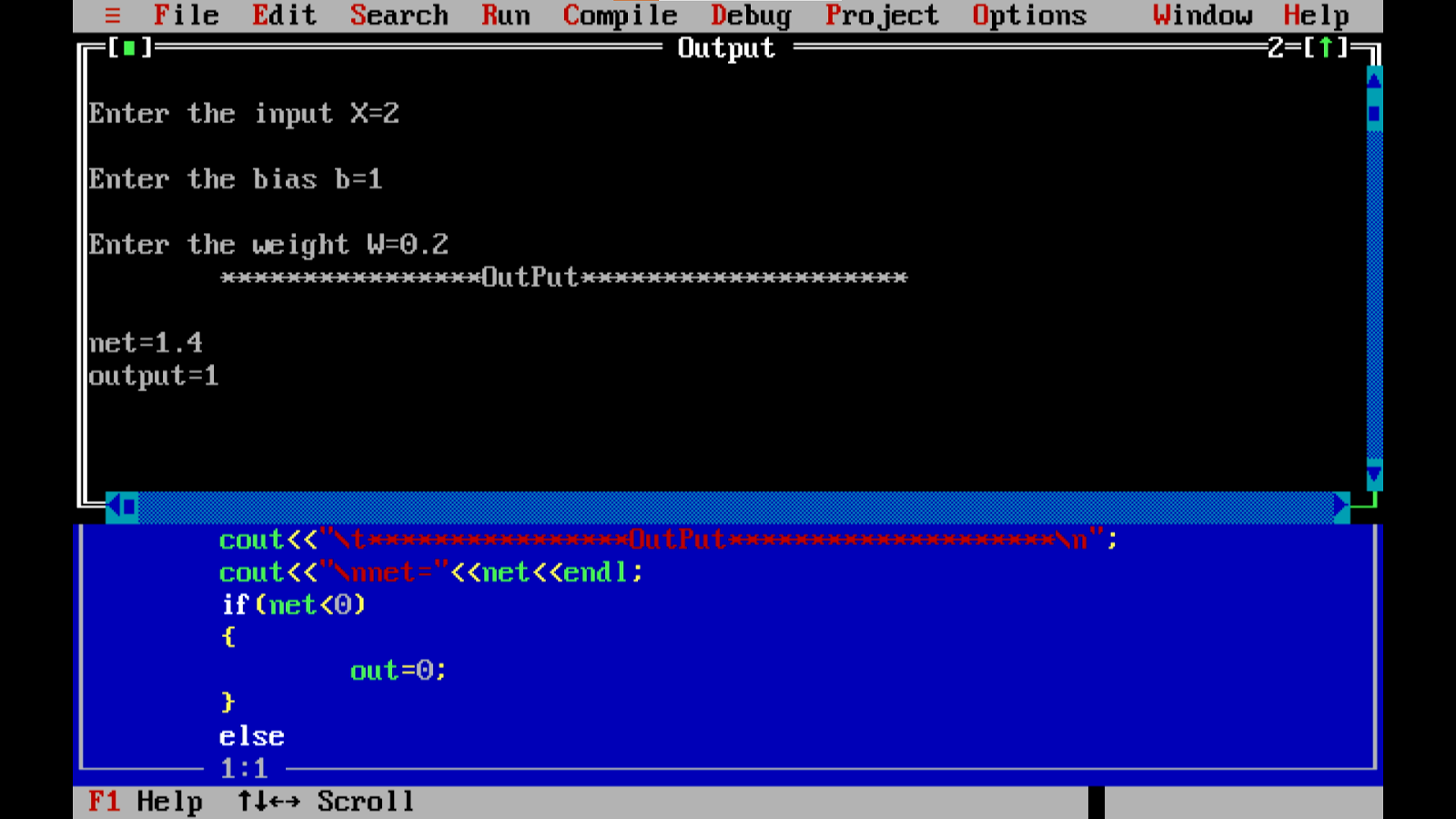
out=1;

cout<<"output="<<out<<endl;

getch();

}

**Output:**



**B) Calculate the output of neural net using both binary and bipolar sigmoidal function.**

# In[1]:

# number of elements as input

n = int(input("Enter number of elements : "))

# In[2]:

print("Enter the inputs")

inputs = [] # creating an empty list for inputs

# iterating till the range

for i in range(0, n):

ele = float(input())

inputs.append(ele) # adding the element

print(inputs)

# In[3]:

print("Enter the weights")

# creating an empty list for weights

weights = []

# iterating till the range

for i in range(0, n):

ele = float(input())

weights.append(ele) # adding the element

print(weights)

# In[4]:

print("The net input can be calculated as   Yin = x1w1 + x2w2 + x3w3")

# In[5]:

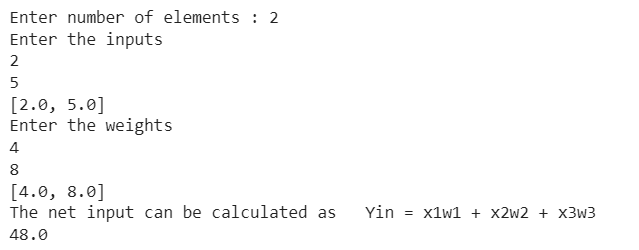
Yin = []

for i in range(0, n):

Yin.append(inputs[i]\*weights[i])

print(round(sum(Yin),3))

**Output:**



# Practical No: 02

**Aim: A) Generate AND/NOT function using McCulloch-Pitts neural net.**

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

w1 = 1

w2 = 1

print("For the ", num\_ip , " inputs calculate the net input using yin = x1w1 + x2w2 ")

x1 = []

x2 = []

for j in range(0, num\_ip):

  ele1 = int(input("x1 = "))

  ele2 = int(input("x2 = "))

  x1.append(ele1)

  x2.append(ele2)

  print("x1 = ",x1)

  print("x2 = ",x2)

  n = x1 \* w1

  m = x2 \* w2

  Yin = []

for i in range(0, num\_ip):

  Yin.append(n[i] + m[i])

  print("Yin = ",Yin)

#Assume one weight as excitatory and the other as inhibitory, i.e.,

  Yin = []

for i in range(0, num\_ip):

  Yin.append(n[i] - m[i])

##  print("After assuming one weight as excitatory and the other as inhibitory Yin = ",Yin)

#From the calculated net inputs, now it is possible to fire the neuron for input (1, 0)

#only by fixing a threshold of 1, i.e., θ ≥ 1 for Y unit.

#Thus, w1 = 1, w2 = -1; θ ≥ 1

  Y=[]

for i in range(0, num\_ip):

  if(Yin[i]>=1):

    ele = 1

    Y.append(ele)

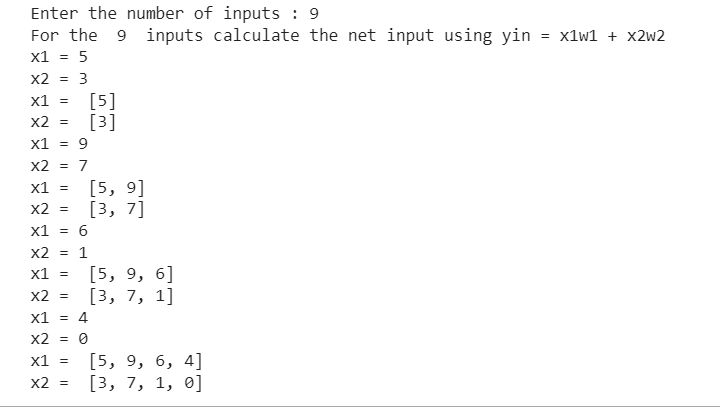
  if(Yin[i]<1):

    ele = 0

    Y.append(ele)

    print("Y = ",Y)

**Output:**



**B) Generate XOR function using McCulloch-Pitts neural net.**

import math

import numpy

import random

# note that this only works for a single layer of depth

INPUT\_NODES = 2

OUTPUT\_NODES = 1

HIDDEN\_NODES = 2

# 15000 iterations is a good point for playing with learning rate

MAX\_ITERATIONS = 130000

# setting this too low makes everything change very slowly, but too high

# makes it jump at each and every example and oscillate. I found .5 to be good

LEARNING\_RATE = .2

print("Neural Network Program")

class network:

  def \_\_init\_\_(self, input\_nodes, hidden\_nodes, output\_nodes, learning\_rate):

    self.input\_nodes = input\_nodes

    self.hidden\_nodes = hidden\_nodes

    self.output\_nodes = output\_nodes

    self.total\_nodes = input\_nodes + hidden\_nodes + output\_nodes

    self.learning\_rate = learning\_rate

        # set up the arrays

    self.values = numpy.zeros(self.total\_nodes)

    self.expectedValues = numpy.zeros(self.total\_nodes)

    self.thresholds = numpy.zeros(self.total\_nodes)

        # the weight matrix is always square

    self.weights = numpy.zeros((self.total\_nodes, self.total\_nodes))

        # set random seed! this is so we can experiment consistently

random.seed(10000)

        # set initial random values for weights and thresholds

        # this is a strictly upper triangular matrix as there is no feedback

        # loop and there inputs do not affect other inputs

    for i in range(self.input\_nodes, self.total\_nodes):

      self.thresholds[i] = random.random() / random.random()

      for j in range(i + 1, self.total\_nodes):

        self.weights[i][j] = random.random() \* 2

  def process(self):

        # update the hidden nodes

    for i in range(self.input\_nodes, self.input\_nodes + self.hidden\_nodes):

            # sum weighted input nodes for each hidden node, compare threshold, apply sigmoid

      W\_i = 0.0

      for j in range(self.input\_nodes):

        W\_i += self.weights[j][i] \* self.values[j]

      W\_i -= self.thresholds[i]

      self.values[i] = 1 / (1 + math.exp(-W\_i))

        # update the output nodes

    for i in range(self.input\_nodes + self.hidden\_nodes, self.total\_nodes):

            # sum weighted hidden nodes for each output node, compare threshold, apply sigmoid

      W\_i = 0.0

      for j in range(self.input\_nodes, self.input\_nodes + self.hidden\_nodes):

        W\_i += self.weights[j][i] \* self.values[j]

      W\_i -= self.thresholds[i]

      self.values[i] = 1 / (1 + math.exp(-W\_i))

  def processErrors(self):

    sumOfSquaredErrors = 0.0

        # we only look at the output nodes for error calculation

    for i in range(self.input\_nodes + self.hidden\_nodes, self.total\_nodes):

      error = self.expectedValues[i] - self.values[i]

            #print error

      sumOfSquaredErrors += math.pow(error, 2)

      outputErrorGradient = self.values[i] \* (1 - self.values[i]) \* error

            #print outputErrorGradient

            # now update the weights and thresholds

      for j in range(self.input\_nodes, self.input\_nodes + self.hidden\_nodes):

                # first update for the hidden nodes to output nodes (1 layer)

        delta = self.learning\_rate \* self.values[j] \* outputErrorGradient

                #print delta

        self.weights[j][i] += delta

        hiddenErrorGradient = self.values[j] \* (1 - self.values[j]) \* outputErrorGradient \* self.weights[j][i]

                # and then update for the input nodes to hidden nodes

        for k in range(self.input\_nodes):

          delta = self.learning\_rate \* self.values[k] \* hiddenErrorGradient

          self.weights[k][j] += delta

                # update the thresholds for the hidden nodes

        delta = self.learning\_rate \* -1 \* hiddenErrorGradient

                #print delta

        self.thresholds[j] += delta

            # update the thresholds for the output node(s)

      delta = self.learning\_rate \* -1 \* outputErrorGradient

      self.thresholds[i] += delta

    return sumOfSquaredErrors

class sampleMaker:

  def \_\_init\_\_(self, network):

    self.counter = 0

    self.network = network

  def setXor(self, x):

    if x == 0:

      self.network.values[0] = 1

      self.network.values[1] = 1

      self.network.expectedValues[4] = 0

    elif x == 1:

      self.network.values[0] = 0

      self.network.values[1] = 1

      self.network.expectedValues[4] = 1

    elif x == 2:

      self.network.values[0] = 1

      self.network.values[1] = 0

      self.network.expectedValues[4] = 1

    else:

      self.network.values[0] = 0

      self.network.values[1] = 0

      self.network.expectedValues[4] = 0

  def setNextTrainingData(self):

    self.setXor(self.counter % 4)

    self.counter += 1

# start of main program loop, initialize classes

net = network(INPUT\_NODES, HIDDEN\_NODES, OUTPUT\_NODES, LEARNING\_RATE)

samples = sampleMaker(net)

for i in range(MAX\_ITERATIONS):

  samples.setNextTrainingData()

  net.process()

  error = net.processErrors()

    # prove that we got the right answers(ish)!

  if i > (MAX\_ITERATIONS - 5):

    output = (net.values[0], net.values[1], net.values[4], net.expectedValues[4], error)

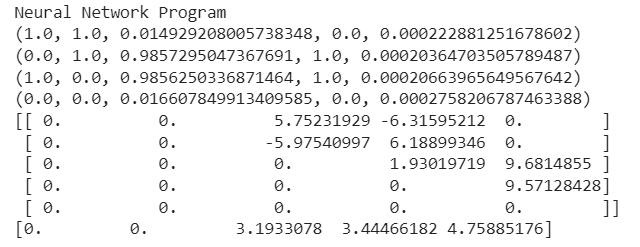
    print(output)

# display final parameters

print(net.weights)

print(net.thresholds)

**Output:**



# Practical No: 03

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

#include<iostream.h>

#include<conio.h>

void main()

{

float n,w,t,net,div,a,al;

cout<<"consider o single neuron perceptron with a single i/p";

cin>>w;

cout<<"enter the learning coefficient";

cin>>d;

for (i=0;i<10;i++)

{

net = x+w;

if(wt<0)

a=0;

else

a=1;

div=at+a+w;

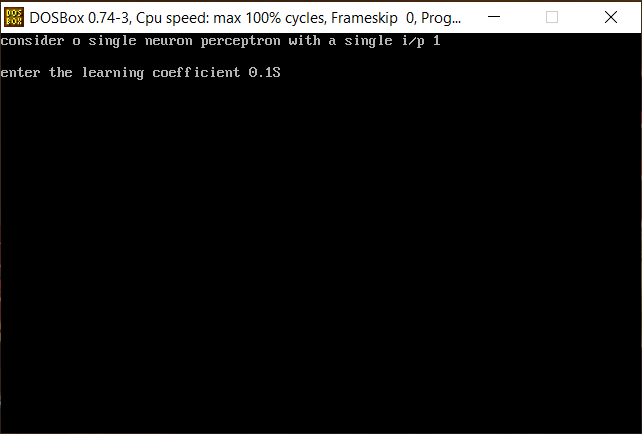
w=w+div;

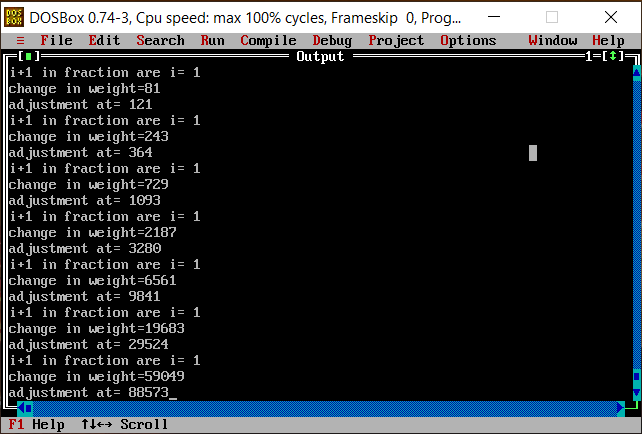
cout<<"i+1 in fraction are i"<<a<<"change in weight"<<dw<<"adjustment at="<<w;

}

}

**Output:**





**B) Write a program to implement of delta rule.**

#supervised learning

import numpy as np

import time

np.set\_printoptions(precision=2)

x=np.zeros((3,))

weights=np.zeros((3,))

desired=np.zeros((3,))

actual=np.zeros((3,))

for i in range(0,3):

x[i]=float(input("Initial inputs:"))

for i in range(0,3):

weights[i]=float(input("Initial weights:"))

for i in range(0,3):

desired[i]=float(input("Desired output:"))

a=float(input("Enter learning rate:"))

actual=x\*weights

print("actual",actual)

print("desired",desired)

while True:

if np.array\_equal(desired,actual):

break #no change

else:

for i in range(0,3):

weights[i]=weights[i]+a\*(desired[i]-actual[i])

actual=x\*weights

print("weights",weights)

print("actual",actual)

print("desired",desired)

print("\*"\*30)

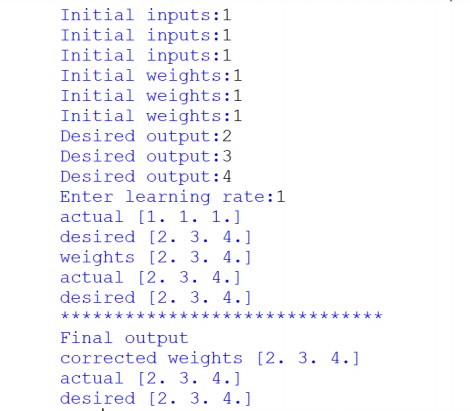
print("Final output")

print("Corrected weights",weights)

print("actual",actual)

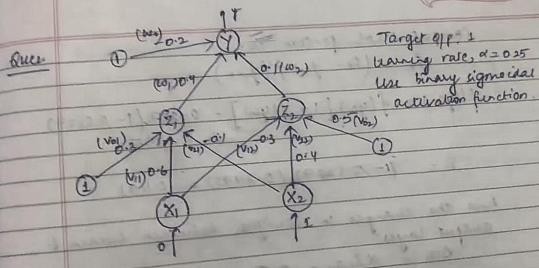
print("desired",desired)

**Output:**



# Practical No: 04

**Aim: A) Write a program for back propagation algorithm.**



import numpy as np import decimal import math

np.set\_printoptions(precision=2) v1=np.array([0.6, 0.3])

v2=np.array([-0.1, 0.4])

w=np.array([-0.2,0.4,0.1]) b1=0.3

b2=0.5 x1=0 x2=1

alpha=0.25

print("calculate net input to z1 layer") zin1=round(b1+ x1\*v1[0]+x2\*v2[0],4) print("z1=",round(zin1,3)) print("calculate net input to z2 layer") zin2=round(b2+ x1\*v1[1]+x2\*v2[1],4) print("z2=",round(zin2,4))

print("Apply activation function to calculate output") z1=1/(1+math.exp(-zin1))

z1=round(z1,4) z2=1/(1+math.exp(-zin2)) z2=round(z2,4) print("z1=",z1)

print("z2=",z2)

print("calculate net input to output layer") yin=w[0]+z1\*w[1]+z2\*w[2]

print("yin=",yin) print("calculate net output")

y=1/(1+math.exp(-yin)) print("y=",y)

fyin=y \*(1- y) dk=(1-y)\*fyin print("dk",dk)

dw1= alpha \* dk \* z1 dw2= alpha \* dk \* z2 dw0= alpha \* dk

print("compute error portion in delta") din1=dk\* w[1]

din2=dk\* w[2] print("din1=",din1) print("din2=",din2) print("error in delta") fzin1= z1 \*(1-z1) print("fzin1",fzin1) d1=din1\* fzin1 fzin2= z2 \*(1-z2) print("fzin2",fzin2) d2=din2\* fzin2 print("d1=",d1)

print("d2=",d2)

print("Changes in weights between input and hidden layer") dv11=alpha \* d1 \* x1

print("dv11=",dv11) dv21=alpha \* d1 \* x2 print("dv21=",dv21) dv01=alpha \* d1 print("dv01=",dv01) dv12=alpha \* d2 \* x1 print("dv12=",dv12) dv22=alpha \* d2 \* x2 print("dv22=",dv22) dv02=alpha \* d2 print("dv02=",dv02)

print("Final weights of network") v1[0]=v1[0]+dv11 v1[1]=v1[1]+dv12

print("v=",v1) v2[0]=v2[0]+dv21 v2[1]=v2[1]+dv22

print("v2",v2)

w[1]=w[1]+dw1

w[2]=w[2]+dw2 b1=b1+dv01 b2=b2+dv02

w[0]=w[0]+dw0

print("w=",w)

print("bias b1=",b1, " b2=",b2)

**Output:**



**B) Write a program for error back propagation algorithm (ebpa)learning.**

import math a0=-1

t=-1

w10=float(input("Enter weight first network")) b10=float(input("Enter base first network:")) w20=float(input("Enter weight second network:")) b20=float(input("Enter base second network:")) c=float(input("Enter learning coefficient:")) n1=float(w10\*c+b10)

a1=math.tanh(n1) n2=float(w20\*a1+b20)

a2=math.tanh(float(n2)) e=t-a2

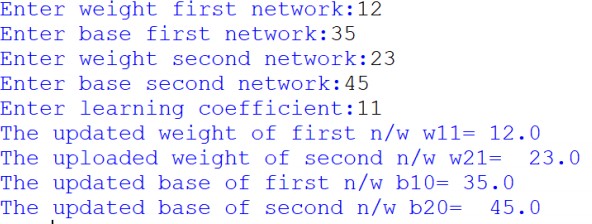
s2=-2\*(1-a2\*a2)\*e s1=(1-a1\*a1)\*w20\*s2 w21=w20-(c\*s2\*a1) w11=w10-(c\*s1\*a0) b21=b20-(c\*s2)

b11=b10-(c\*s1)

print("The updated weight of first n/w w11=",w11) print("The uploaded weight of second n/w w21= ",w21)

print("The updated base of first n/w b10=",b10) print("The updated base of second n/w b20= ",b20)

**Output:**



# Practical No: 05

**Aim: A) Write a program for hopfield network.**

import numpy as np

class HopfieldNetwork:

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

self.weights = np.zeros((len(patterns[0]), len(patterns[0])))

for pattern in patterns:

self.weights += np.outer(pattern, pattern)

np.fill\_diagonal(self.weights, 0)

def update\_neuron(self, neuron\_index, state):

return np.sign(np.dot(self.weights[neuron\_index], state))

def update\_state(self, state):

for i in range(len(state)):

state[i] = self.update\_neuron(i, state)

return state

def set\_initial\_state(self, state):

self.current\_state = state

def get\_current\_state(self):

return self.current\_state

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

patterns = [[1, 1, -1, -1], [-1, 1, -1, 1]]

network = HopfieldNetwork(patterns)

initial\_state = [1, -1, -1, 1]

network.set\_initial\_state(initial\_state)

for \_ in range(10):

network.update\_state(network.get\_current\_state())

final\_state = network.get\_current\_state()

print("Final state of the network: ", final\_state)

**Output:**

Final state of the network: [-1.0, 1.0, -1.0, 1.0]

**B) Write a program for radial basis function.**

from scipy import \*

from scipy.linalg import norm, pinv from matplotlib import pyplot as plt class RBF:

def init (self, indim, numCenters, outdim): self.indim =indim

self.outdim =outdim self.numCenters =numCenters

self.centers =[random.uniform(-1, 1, indim) for i in range(numCenters)] self.beta =8

self.W =random.random((self.numCenters, self.outdim)) def \_basisfunc(self, c, d):

assert len(d) ==self.indim

return exp(-self.beta \*norm(c-d)\*\*2) def \_calcAct(self, X):

# calculate activations of RBFs

G =zeros((X.shape[0], self.numCenters), float) for ci, c in enumerate(self.centers):

for xi, x in enumerate(X):

G[xi,ci] =self.\_basisfunc(c, x) return G

def train(self, X, Y):

""" X: matrix of dimensions n x indim y: column vector of dimension n x 1 """

# choose random center vectors from training set

rnd\_idx =random.permutation(X.shape[0])[:self.numCenters] self.centers =[X[i,:] for i in rnd\_idx]

print("center", self.centers)

# calculate activations of RBFs G =self.\_calcAct(X)

print (G)

# calculate output weights (pseudoinverse) self.W =dot(pinv(G), Y)

def test(self, X):

""" X: matrix of dimensions n x indim """ G =self.\_calcAct(X)

Y =dot(G, self.W) return Y

if name ==' main ':

# 1D Example n =100

x =mgrid[-1:1:complex(0,n)].reshape(n, 1) # set y and add random noise

y =sin(3\*(x+0.5)\*\*3-1)

# y += random.normal(0, 0.1, y.shape) # rbf regression

rbf =RBF(1, 10, 1)

rbf.train(x, y) z =rbf.test(x)

# plot original data plt.figure(figsize=(12, 8)) plt.plot(x, y, 'k-')

# plot learned model plt.plot(x, z, 'r-', linewidth=2) # plot rbfs

plt.plot(rbf.centers, zeros(rbf.numCenters), 'gs') for c in rbf.centers:

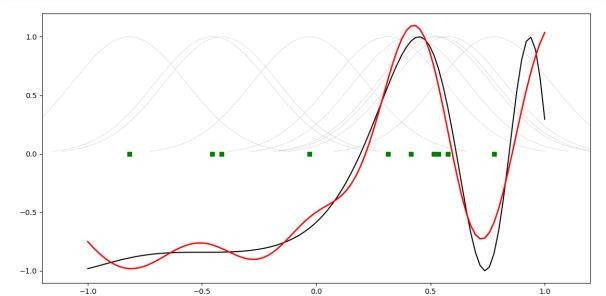
# RF prediction lines

cx =arange(c-0.7, c+0.7, 0.01)

cy =[rbf.\_basisfunc(array([cx\_]), array([c])) for cx\_ in cx] plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

plt.xlim(-1.2, 1.2) plt.show()

**Output:**

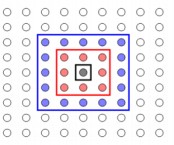


# Practical No: 06

**Aim: A) Self-Organizing Maps.**

The SOM algorithm is used to compress the information to produce a similarity graph while preserving the topologic relationship of the input data space. The basic SOM model construction algorithm can be interpreted as follows: 1) Create and initialize a matrix (weight vector) randomly to hold the neurons. If the matrix can be initialized with order and roughly compiles with the input density function, the map will converge quickly 2) Read the input data space. For each observation (instance), use the optimum fit approach, which is based on the Euclidean distance c = 𝑎𝑟𝑔𝑖 min || x – 𝑚𝑖 || to find the neuron which best matches this observation. Let x denote the training vector from the observation and 𝑚𝑖 denote a single neuron in the matrix. Update that neuron to resemble that observation using the following equation: 𝑚𝑖 (𝑡 + 1) = 𝑚𝑖 (𝑡) + ℎ(𝑡)[𝑥(𝑡) − 𝑚𝑖 (𝑡)] (4) 𝑚𝑖(𝑡): the weight vector before the neuron is updated. (𝑡 + 1): the weight vector after the neuron is updated. (𝑡): the training vector from the observation. ℎ(𝑡): the neighborhood function (a smoothing kernel defined over the lattice points), defined though the following equation: ℎ(𝑡) = { 𝛼(𝑡), 𝑖∈𝑁𝑐 0, 𝑖∈𝑁𝑐

(5) : the neighborhood set, which decreases with time. (𝑡): the learning-rate factor which can be linear, exponential or inversely proportional. It is a monotonically decreasing function of time (t)



In general, SOMs might be useful for visualizing high-dimensional data in terms of its similarity structure. Especially large SOMs (i.e. with large number of Kohonen units) are known to perform mappings that preserve the topology of the original data, i.e. neighboring data points in input space will also be represented in adjacent locations on the SOM. The following code shows the ‘classic’ color mapping example, i.e. the SOM will map a number of colors into a rectangular area.

## frommvpa2.suiteimport\*

First, we define some colors as RGB values from the interval (0,1), i.e. with white being (1, 1, 1) and black being (0, 0, 0). Please note, that a substantial proportion of the defined colors represent variations of ‘blue’, which are supposed to be represented in more detail in the SOM.

colors=np.array( [[0.,0.,0.],

[0.,0.,1.],

[0.,0.,0.5],

[0.125,0.529,1.0],

[0.33,0.4,0.67],

[0.6,0.5,1.0],

[0.,1.,0.],

[1.,0.,0.],

[0.,1.,1.],

[1.,0.,1.],

[1.,1.,0.],

[1.,1.,1.],

[.33,.33,.33],

[.5,.5,.5],

[.66,.66,.66]])

# store the names of the colors for visualization later on color\_names= \

['black','blue','darkblue','skyblue',

'greyblue','lilac','green','red',

'cyan','violet','yellow','white', 'darkgrey','mediumgrey','lightgrey']

Now we can instantiate the mapper. It will internally use a so-called Kohonen layer to map the data onto. We tell the mapper to use a rectangular layer with 20 x 30 units. This will be the output space of the mapper. Additionally, we tell it to train the network using 400 iterations and to use custom learning rate.

som=SimpleSOMMapper((20,30),400,learning\_rate=0.05)

Finally, we train the mapper with the previously defined ‘color’ dataset. som.train(colors)

Each unit in the Kohonen layer can be treated as a pointer into the high-dimensional input space, that can be queried to inspect which input subspaces the SOM maps onto certain sections of its 2D output space. The color-mapping generated by this example’s SOM can be shown with a single matplotlib call:

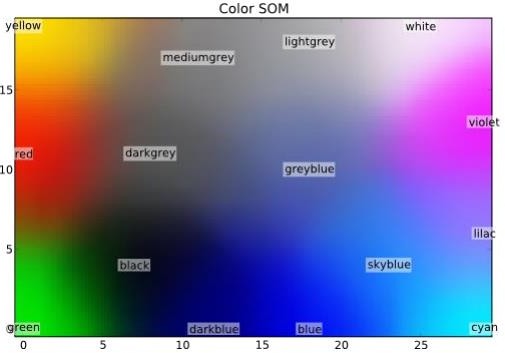
pl.imshow(som.K,origin='lower')

And now, let’s take a look onto which coordinates the initial training prototypes were mapped to. The get those coordinates we can simply feed the training data to the mapper and plot the output

mapped=som(colors) pl.title('Color SOM')

# SOM's kshape is (rows x columns), while matplotlib wants (X x Y) fori,minenumerate(mapped): pl.text(m[1],m[0],color\_names[i],ha='center',va='center', bbox=dict(facecolor='white',alpha=0.5,lw=0))

**Output:**



**B) Adaptive Resonance Theory**

import numpy as np

class ART:

    def \_\_init\_\_(self, vigilance\_parameter, input\_vectors):

        self.vigilance\_parameter = vigilance\_parameter

        self.input\_vectors = input\_vectors

        self.prototypes = []

    def compare\_vectors(self, input\_vector, prototype):

        match = np.dot(input\_vector, prototype) / (np.linalg.norm(input\_vector) \* np.linalg.norm(prototype))

        return match

    def create\_prototype(self, input\_vector):

        return input\_vector / np.linalg.norm(input\_vector)

    def compare\_with\_prototypes(self, input\_vector):

        max\_match = 0

        max\_match\_index = -1

        for index, prototype in enumerate(self.prototypes):

            match = self.compare\_vectors(input\_vector, prototype)

            if match > max\_match:

                max\_match = match

                max\_match\_index = index

        return max\_match, max\_match\_index

    def adapt\_prototype(self, input\_vector, prototype):

        return prototype + (input\_vector - prototype)

    def train(self):

        for input\_vector in self.input\_vectors:

            match, match\_index = self.compare\_with\_prototypes(input\_vector)

            if match < self.vigilance\_parameter:

                self.prototypes.append(self.create\_prototype(input\_vector))

            else:

                self.prototypes[match\_index] = self.adapt\_prototype(input\_vector, self.prototypes[match\_index])

    def classify(self, input\_vector):

        match, match\_index = self.compare\_with\_prototypes(input\_vector)

        if match < self.vigilance\_parameter:

            return None

        else:

            return match\_index

if \_\_name\_\_ == "\_\_main\_\_":

    input\_vectors = [[0.5, 0.5], [0.6, 0.6], [0.7, 0.7], [0.8, 0.9], [0.9, 0.8]]

    vigilance\_parameter = 0.8

    art = ART(vigilance\_parameter, input\_vectors)

    art.train()

    result = art.classify([0.6, 0.6])

    if result is None:

        print("Input vector does not match any prototype.")

    else:

        print("Input vector belongs to cluster", result)

**Output:**

Input vector belongs to cluster 0

# Practical No: 07

**Aim: A) Line Separation.**

import numpy as np

import matplotlib.pyplot as plt

def create\_distance\_function(a, b, c):

""" 0 = ax + by + c """ def distance(x, y):

""" returns tuple (d, pos) d is the distance

If pos == -1 point is below the line, 0 on the line and +1 if above the line """

nom = a \* x + b \* y + c if nom == 0:

pos = 0

elif (nom<0 and b<0) or (nom>0 and b>0): pos = -1

else:

pos = 1

return (np.absolute(nom) / np.sqrt( a \*\* 2 + b \*\* 2), pos) return distance

points = [ (3.5, 1.8), (1.1, 3.9) ]

fig, ax = plt.subplots() ax.set\_xlabel("sweetness") ax.set\_ylabel("sourness") ax.set\_xlim([-1, 6])

ax.set\_ylim([-1, 8])

X = np.arange(-0.5, 5, 0.1)

colors = ["r", ""] # for the samples size = 10

for (index, (x, y)) in enumerate(points): if index== 0:

ax.plot(x, y, "o", color="darkorange", markersize=size) else:

ax.plot(x, y, "oy", markersize=size) step = 0.05

for x in np.arange(0, 1+step, step):

slope = np.tan(np.arccos(x))

dist4line1 = create\_distance\_function(slope, -1, 0) #print("x: ", x, "slope: ", slope)

Y = slope \* X results = []

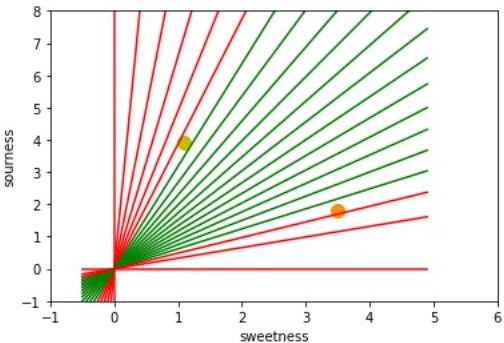
for point in points: results.append(dist4line1(\*point)) #print(slope, results)

if (results[0][1] != results[1][1]):

ax.plot(X, Y, "g-") else:

ax.plot(X, Y, "r-") plt.show()

**Output:**



# Practical No: 08

**Aim: A) Membership and Identity Operators IN, NOT IN.**

# Membership operator 'in'

def check\_membership(value, container):

if value in container:

print(value, "is in the container.")

else:

print(value, "is not in the container.")

# Membership operator 'not in'

def check\_not\_membership(value, container):

if value not in container:

print(value, "is not in the container.")

else:

print(value, "is in the container.")

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

container = [1, 2, 3, 4, 5]

value = 3

check\_membership(value, container) # 3 is in the container.

check\_not\_membership(6, container) # 6 is not in the container.

**Output:**

3 is in the container.

6 is not in the container.

**B) Membership and Identity Operators IS, IS NOT.**

# Identity operator 'is'

def check\_identity(value1, value2):

    if value1 is value2:

        print(value1, "and", value2, "have the same identity.")

    else:

        print(value1, "and", value2, "do not have the same identity.")

# Identity operator 'is not'

def check\_not\_identity(value1, value2):

    if value1 is not value2:

        print(value1, "and", value2, "do not have the same identity.")

    else:

        print(value1, "and", value2, "have the same identity.")

if \_\_name\_\_ == "\_\_main\_\_":

    value1 = [1, 2, 3]

    value2 = [1, 2, 3]

    check\_identity(value1, value2) # [1, 2, 3] and [1, 2, 3] do not have the same identity.

    check\_not\_identity(value1, value2) # [1, 2, 3] and [1, 2, 3] do not have the same identity.

    value2 = value1

    check\_identity(value1, value2) # [1, 2, 3] and [1, 2, 3] have the same identity.

    check\_not\_identity(value1, value2) # [1, 2, 3] and [1, 2, 3] have the same identity.

**Output:**

[1, 2, 3] and [1, 2, 3] do not have the same identity.

[1, 2, 3] and [1, 2, 3] do not have the same identity.

[1, 2, 3] and [1, 2, 3] have the same identity.

[1, 2, 3] and [1, 2, 3] have the same identity.