**Practical No: 01**

A)Design a simple linear neural network model.

x=float(input("Enter value of x:"))

w=float(input("Enter value of weight w:"))

b=float(input("Enter value of bias b:"))

net = int(w\*x+b)

if(net<0):

  out=0

elif((net>=0)&(net<=1)):

  out =net

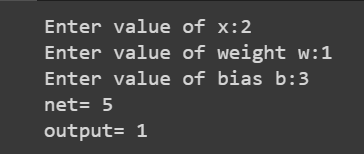
else:

  out=1

print("net=",net)

print("output=",out)

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]:

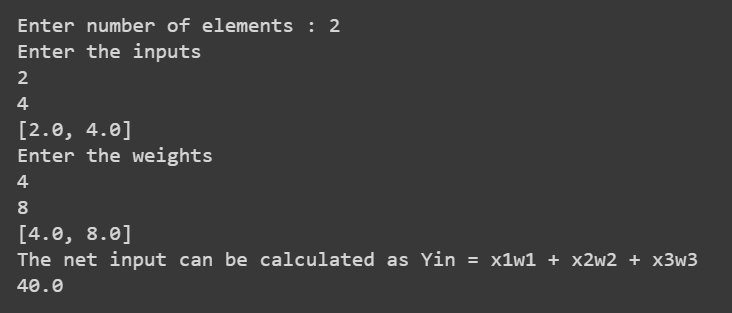
Yin = []

for i in range(0, n):

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

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

output:



**Practical No: 02**

**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

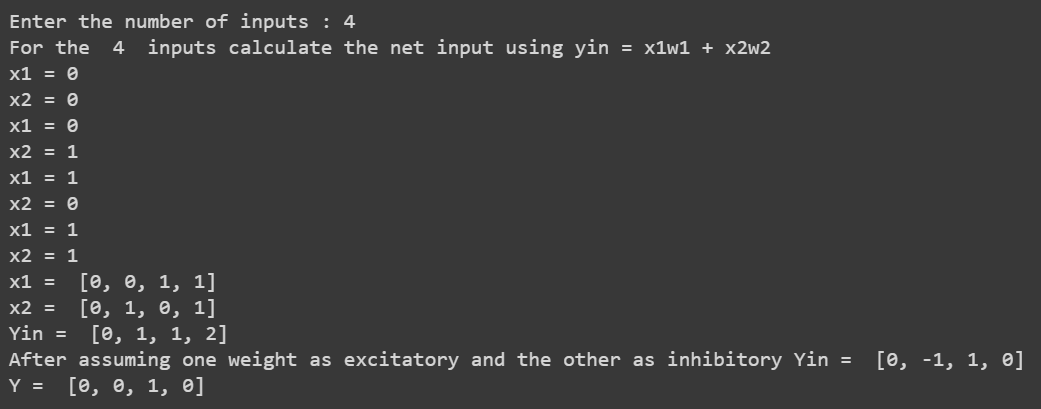
    Y.append(ele)

  if(Yin[i]<1):

    ele = 0

    Y.append(ele)

print("Y = ",Y)

**output: **

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

import numpy as np

print('Enter weights')

w11=int(input('Weight w11='))

w12=int(input('weight w12='))

w21=int(input('Weight w21='))

w22=int(input('weight w22='))

v1=int(input('weight v1='))

v2=int(input('weight v2='))

print('Enter Threshold Value')

theta=int(input('theta='))

x1=np.array([0, 0, 1, 1])

x2=np.array([0, 1, 0, 1])

z=np.array([0, 1, 1, 0])

con=1

y1=np.zeros((4,))

y2=np.zeros((4,))

y=np.zeros((4,))

while con==1:

  zin1=np.zeros((4,))

zin2=np.zeros((4,))

zin1=x1\*w11+x2\*w21

zin2=x1\*w21+x2\*w22

print("z1",zin1)

print("z2",zin2)

for i in range(0,4):

  if zin1[i]>=theta:

    y1[i]=1

  else:

    y1[i]=0

if zin2[i]>=theta:

    y2[i]=1

else:

    y2[i]=0

yin=np.array([])

yin=y1\*v1+y2\*v2

for i in range(0,4):

  if yin[i]>=theta:

    y[i]=1

else:

  y[i]=0

print("yin",yin)

print('Output of Net')

y=y.astype(int)

print("y",y)

print("z",z)

if np.array\_equal(y,z):

  con=0

else:

  print("Net is not learning enter another set of weights and Threshold value")

w11=input("Weight w11=")

w12=input("weight w12=")

w21=input("Weight w21=")

w22=input("weight w22=")

v1=input("weight v1=")

v2=input("weight v2=")

theta=input("theta=")

print("McCulloch-Pitts Net for XOR function")

print("Weights of Neuron Z1")

print(w11)

print(w21)

print("weights of Neuron Z2")

print(w12)

print(w22)

print("weights of Neuron Y")

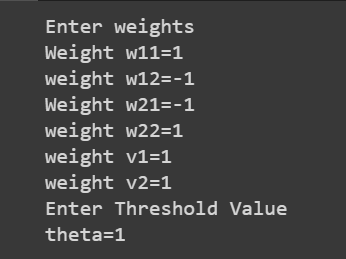
print(v1)

print(v2)

print("Threshold value")

print(theta)

**output:**

****

**Practical No: 03**

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

import numpy as np

#first pattern

x1=np.array([1,1,1,-1,1,-1,1,1,1])

#second pattern

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

#initialize bais value

b=0

#define target

y=np.array([1,-1])

wtold=np.zeros((9,))

wtnew=np.zeros((9,))

wtnew=wtnew.astype(int)

wtold=wtold.astype(int)

bais=0

print("First input with target =1")

for i in range(0,9):

  wtold[i]=wtold[i]+x1[i]\*y[0]

wtnew=wtold

b=b+y[0]

print("new wt =", wtnew)

print("Bias value",b)

print("Second input with target =-1")

for i in range(0,9):

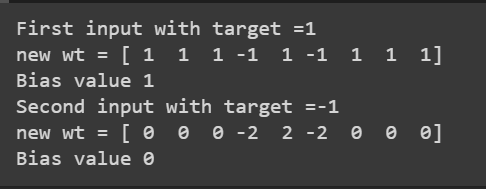
  wtnew[i]=wtold[i]+x2[i]\*y[1]

b=b+y[1]

print("new wt =", wtnew)

print("Bias value",b)

**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)

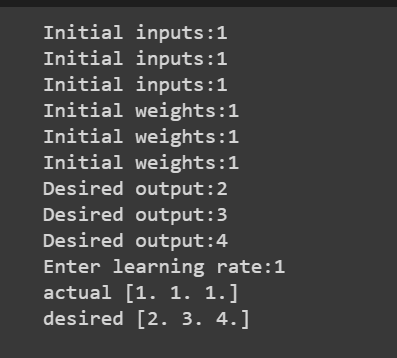
print("Final output")

print("Corrected weights",weights)

print("actual",actual)

print("desired",desired)

**output:**

****

**Practical No: 04**

**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)

**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)

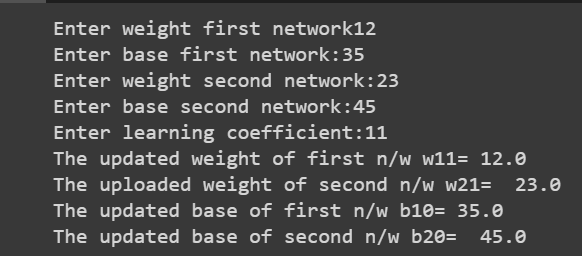
print("The updated weight of first n/w w11=",w11)

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**

**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)

**Practical No: 06**

**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)

**Practical No: 08**

**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.

**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.