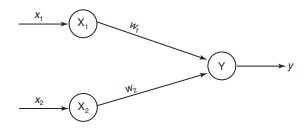
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INDEX** | | | | |
| **Sr. No.** | **Topic** | **Date** | **Page No.** | **Sign** |
| 1. | **Implement the following** |  |  |  |
| a. | Design a simple linear neural network model | 24/09/22 | 1 |  |
| b. | Calculate the output of neural net using both binary and bipolar sigmoidal function | 29/09/22 | 3 |  |
| 2. | **Implement the following** |  |  |  |
| a. | Generate AND/NOT function using McCulloch-Pitts neural net. | 1/10/22 | 5 |  |
| b. | Generate XOR function using McCulloch-Pitts neural net. | 06/10/22 | 7 |  |
| 3. | **Implement the following** |  |  |  |
| a. | Write a program to implement Hebb’s rule. | 15/10/22 | 9 |  |
| b. | Write a program to implement of delta rule. | 20/10/22 | 11 |  |
| 4. | **Implement the following** |  |  |  |
| a. | Write a program for Back Propagation Algorithm | 12/11/22 | 13 |  |
| b. | Write a program for error Backpropagation algorithm. | 17/11/22 | 15 |  |
| 5. | **Implement the following** |  |  |  |
| a. | Write a program for Hopfield Network. | 19/11/22 | 18 |  |
| 6. | **Implement the following** |  |  |  |
| a. | Write a program for Linear separation. | 24/11/22 | 21 |  |
| 7. | **Implement the following** |  |  |  |
| a. | Membership and Identity Operators | in, not in, | 26/11/22 | 23 |  |
| b. | Membership and Identity Operators is, is not | 01/12/22 | 24 |  |
| 8. | **Implement the following** |  |  |  |
| a. | Find ratios using fuzzy logic | 03/12/22 | 25 |  |
| 9. | **Implement the following** |  |  |  |
| a. | Implementation of Simple genetic algorithm | 10/12/22 | 26 |  |

**Practical 1**

**A. Design a simple linear neural network model**

**Theory:** A neural network is a processing device, either an algorithm or an actual hardware, whose design was inspired by the design and functioning of animal brains and components thereof. The computing world has a lot to gain from neural networks, also known as artificial neural networks or neural net. Artificial neural networks are inspired by the biological neurons within the human body which activate under certain circumstances resulting in a related action performed by the body in response.

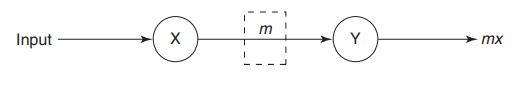
An artificial neural network (ANN) is an efficient information processing system which resembles in characteristics with a biological neural network. ANNs possess large number of highly interconnected processing elements called nodes or units or neurons, which usually operate in parallel and are configured in regular architectures. Each neuron is connected with the other by a connection link. Each connection link is associated with weights which contain information about the input signal. This information is used by the neuron net to solve a particular problem. ANNs’ collective behaviour is characterized by their ability to learn, recall and generalize training patterns or data similar to that of a human brain. They have Each neuron receives a multiplied version of inputs and random weights, which is then added with a static bias value (unique to each neuron layer), this is then passed to an appropriate activation function which decides the final value to be given out of the neuron.



Here X1 and X2 are input neurons, which transmit signals, and Y is the output neuron, which receives signals. Input neurons X1 and X2 are connected to the output neuron Y, over a weighted interconnection links (W1 and W2 ). The bias included in the network has its impact in calculating the net input. The net input has to be calculated in the following way:



The output y of the output neuron Y can be obtained by applying activations over the net input, the activation function is applied over yin to calculate the output. i.e., the function of the net input: y=f(yin) i.e. Output = Function (net input calculated). The neural net of a pure linear equation is (y = mx)



**Code:**

# Prac 1A Design a simple neural network model.

print('--Practical 1A - Design simple linear neural network model--')

x=float(input("Enter Input for Linear Neural Network: "))

weight=float(input("Enter Weight for Linear Neural Network: "))

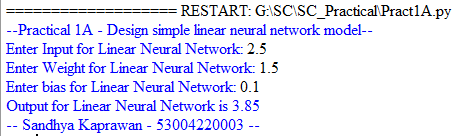
bias=float(input("Enter bias for Linear Neural Network: "))

yin=(x\*weight)+bias

print("Output for Linear Neural Network is "+ str(yin))

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**



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

**Theory:** The function to be applied over the net input is called activation function. This activation helps in achieving the exact output. In a similar way, the activation function is applied over the net input to calculate the output of an ANN. Certain nonlinear functions are used to achieve the advantages of a multilayer network from a single-layer network. When a signal is fed through a multilayer network with linear activation functions, the output obtained remains same as that could be obtained using a single-layer network. Due to this reason, nonlinear functions are widely used in multilayer networks compared to linear functions.

Some of the few activation function are:

1. Identity function: It is a linear function and can be defined as f(x)=x for all x

2. Binary step function: This function can be defined as f(x) = {1 if x>=theta or 0 if x<theta} where theta represents the threshold.

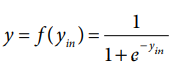
3. Bipolar step function: This function can be defined as f(x) = {1 if x>=theta or -1 if x<theta}

This function is also used in single-layer nets to convert the net input to an output that is bipolar (+1 or –1)

4. Sigmoidal functions: The sigmoidal functions are widely used in back-propagation nets because of the relationship between the value of the functions at a point and the value of the derivative at that point which reduces the computational burden during training.

Sigmoidal functions are of two types:

a. Binary sigmoid function: It is also termed as logistic sigmoid function or unipolar sigmoid function. It can be defined as



b. Bipolar sigmoid function: If the network uses a binary data, it is better to convert it to bipolar form and use the bipolar sigmoidal activation function/ This function is defined as:



**Code:**

# Prac 1B Calculate the output of neural net using both binary and bipolar sigmoidal function.

import math

print('--Practical 1B - Calculate the output of neural net using both binary and bipolar sigmoidal function--')

x1=float(input("Enter Input 1 for Linear Neural Network: "))

x2=float(input("Enter Input 2 for Linear Neural Network: "))

w1=float(input("Enter Weight 1 for Linear Neural Network: "))

w2=float(input("Enter Weight 2 for Linear Neural Network: "))

bias=float(input("Enter bias for Linear Neural Network: "))

yin=(x1\*w1)+(x2\*w2)+bias

print(yin)

if yin>0:

print("Binary output is ",1)

else:

print("Binary output is ",0)

yin=(2/(1+math.exp(-yin)))-1

print(yin)

if yin>0:

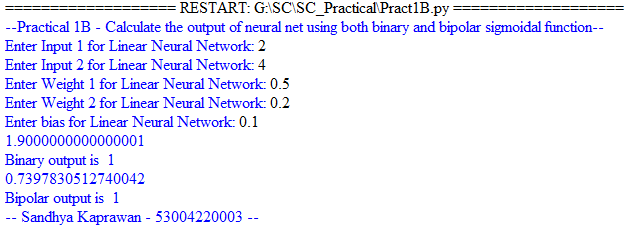
print("Bipolar output is ",1)

else:

print("Bipolar output is",-1)

print("-- Sandhya Kaprawan - 53004220003 --")

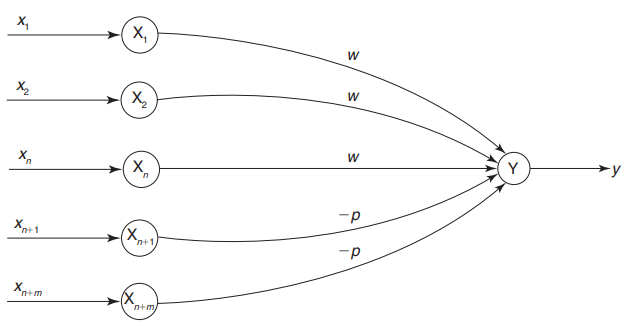
**Output:**



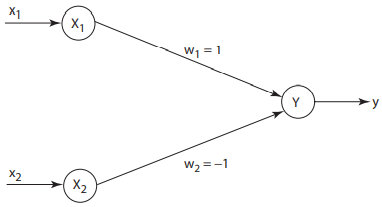
**Practical 2**

**A. Generate ANDNOT function using McCulloch-Pitts neural net**

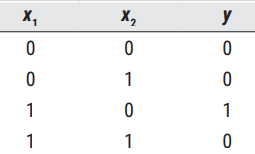
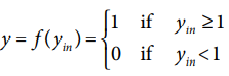
**Theory:** The McCulloch–Pitts neuron was the earliest neural network discovered in 1943. It is usually called the M–P neuron. The M–P neurons are connected by directed weighted paths. It should be noted that the activation of a M–P neuron is binary, that is, at any time step the neuron may fire or may not fire. The weights associated with the communication links may be excitatory (weight is positive) or inhibitory (weight is negative). All the excitatory connected weights entering into a particular neuron will have same weights. The threshold plays a major role in M–P neuron. There is a fixed threshold for each neuron, and if the net input to the neuron is greater than the threshold then the neuron fires. Also, it should be noted that any nonzero inhibitory input would prevent the neuron from firing. The M–P neurons are most widely used in the case of logic functions. Below diagram shows simple representation of McCulloch-Pitts neural network.



Diagrammatic representation of ANDNOT function is as follow:



In the case of ANDNOT function, the response is true if the first input is true and the second input is false. For all other input variations, the response is false. The truth table and Activation function for ANDNOT function is as follow:

**Code:**

print('--Practical 2A - Generate AND/NOT function using McCulloch-Pitts neural net--')

x1=[]

x2=[]

w1=1

w2=-1

y=[]

ycal=[]

yout=[0,0,1,0]

for i in range(0,4):

x=int(input("Enter x1 : "))

x1.append(x)

y1=int(input("Enter x2 : "))

x2.append(y1)

print('x1=',x1)

print('x2=',x2)

print('w1=',w1)

print('w2=',w2)

for i in range(0,4):

y.append(x1[i]\*w1+x2[i]\*w2)

print('Y',y)

for i in range(0,4):

if y[i]>=1:

ycal.append(1)

else:

ycal.append(0)

print('Y calculated:',ycal)

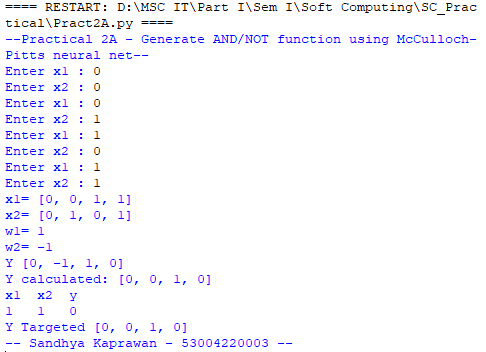
print("x1", " x2", " y")

print(x1[i]," " ,x2[i]," " , y[i])

print('Y Targeted',yout)

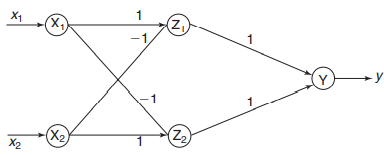
print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**



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

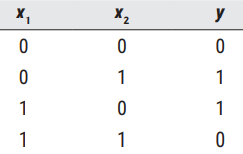
**Theory:** In this case of XOR function, the output is “ON” for only odd number of 1’s. For the rest it is “OFF”. XOR function cannot be represented by single layer neural network an intermediate layer is necessary.



XOR function can be represented as



The truth table for XOR function is as follow:



**Code:**

print('--Practical 2B - Generate XOR function using McCulloch-Pitts neural net--')

x1=[] #0011

x2=[] #0101

x1c=[]

x2c=[]

z1in=[]

z2in=[]

z1out=[]

z2out=[]

yin=[]

ycal=[]

yout=[0,1,1,0]

w1=1

w2=1

for i in range(0,4):

x=int(input("Enter x1 : "))

x1.append(x)

y1=int(input("Enter x2 : "))

x2.append(y1)

print(x1)

print(x2)

for i in range (0,4):

if x1[i]==0:

x1c.append(1)

else:

x1c.append(0)

for i in range (0,4):

if x2[i]==0:

x2c.append(1)

else:

x2c.append(0)

print('x1=',x1)

print('x2=',x2)

print('After Compliment x1:',x1c)

print('After Compliment x2:',x2c)

print('w1=',w1)

print('w2=',w2)

for i in range(0,4):

z1in.append(x1[i]\*w1 + x2c[i]\*w2)

if z1in[i]>1:

z1out.append(1)

else:

z1out.append(0)

print("z1out=", z1out)

for i in range(0,4):

z2in.append(x1c[i]\*w1 + x2[i]\*w2)

if z2in[i]>1:

z2out.append(1)

else:

z2out.append(0)

print("z2out=",z2out)

for i in range(0,4):

yin.append(z1out[i]\*w1 + z2out[i]\*w2)

if yin[i]>=1:

ycal.append(1)

else:

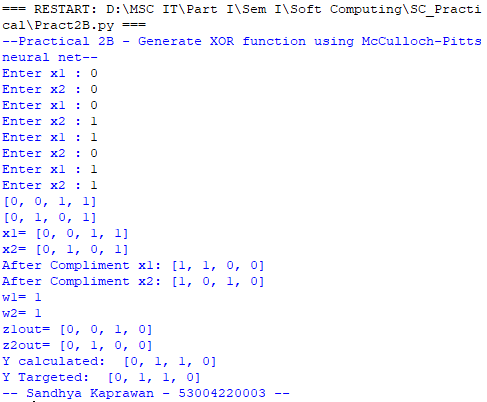
ycal.append(0)

print('Y calculated: ',ycal)

print('Y Targeted: ', yout)

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**



**Practical 3**

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

**Theory:** For a neural net, the Hebb learning rule is a simple one. According to the Hebb rule, the weight vector is found to increase proportionately to the product of the input and the learning signal. Here the learning signal is equal to the neuron’s output. In Hebb learning, if two interconnected neurons are ‘on’ simultaneously then the weights associated with these neurons can be increased by the modification made in their synaptic gap (strength). The weight update and bias update in Hebb rule is given by



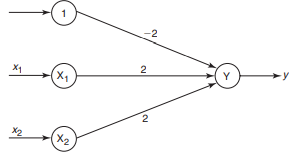
The Hebb rule is more suited for bipolar data than binary data. If binary data is used, the above weight updating formula cannot distinguish two conditions namely:

1. The training pair in which an input unit is “on” and target value is “off”.

2. The training pair in which both the input unit and the target value are “off”.

Thus, there are limitations in Hebb rule application over binary data. Hence, the representation using bipolar data is advantageous.

Hebb’s Neural Network can be represented as:



**Code:**

def hebbian\_learning(samples):

w1,w2,b=0.1,0.1,0.1

for x1, x2, y in samples:

w1=w1+x1\*y

w2=w2+x2\*y

b=b+y

print('Updated weights - w1 =',w1,w2)

print('new bias',b)

AND\_Samples=[

[1,1,1],

[1,0,0],

[0,1,0],

[0,0,0] ]

OR\_Samples=[

[1,1,1],

[1,0,1],

[0,1,1],

[0,0,0] ]

XOR\_Samples=[

[1,1,0],

[1,0,1],

[0,1,1],

[0,0,0] ]

print('AND Sample')

hebbian\_learning(AND\_Samples)

print('OR Sample')

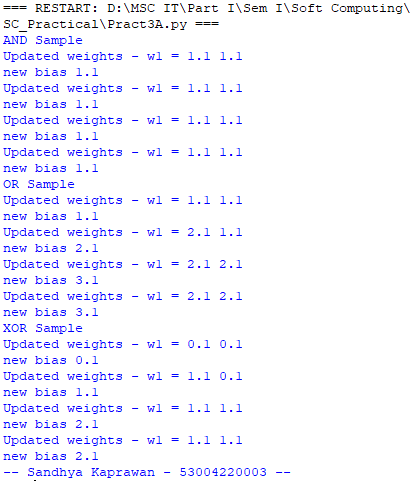
hebbian\_learning(OR\_Samples)

print('XOR Sample')

hebbian\_learning(XOR\_Samples)

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**



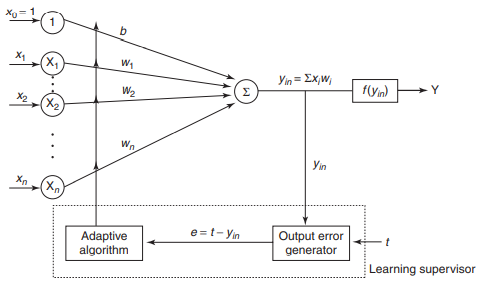
**B. Write a program to implement of Delta rule**

**Theory:** The units with linear activation function are called linear units. A network with a single linear unit is called an Adaline (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable.

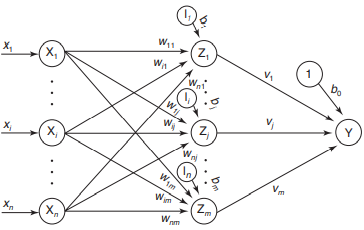
The Widrow-Hoff rule is very similar to perceptron learning rule. However, their origins are different. The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient-descent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of learning steps, but the gradient-descent approach continues forever, converging only asymptotically to the solution. The delta rule updates the weights between the connections so as to minimize the difference between the net input to the output unit and the target value. The major aim is to minimize the error over all training patterns. This is done by reducing the error for each pattern, one at a time. The delta rule for adjusting the weight of ith pattern (i = 1 to n) is

Simple Architecture of Adaline Neural Network is as follow:



The multiple adaptive linear neurons (Madaline) model consists of many Adalines in parallel with a single output unit whose value is based on certain selection rules. It may use majority vote rule. On using this rule, the output would have as answer either true or false. On the other hand, if AND rule is used, the output is true if and only if both the inputs are true, and so on. The weights that are connected from the Adaline layer to the Madaline layer are fixed, positive and possess equal values. The weights between the input layer and the Adaline layer are adjusted during the training process. The Adaline and Madaline layer neurons have a bias of excitation “1’’ connected to them. The training process for a Madaline system is similar to that of an Adaline. Simple Madaline architecture is as follow:



**Code:**

def delta\_rule(samples):

w1,w2,b,alpha=0.1,0.1,0.1,0.2

for x1, x2, t in samples:

yin=x1\*w1+x2\*w2

w1=w1+alpha\*(t-yin)\*x1

w2=w2+alpha\*(t-yin)\*x2

b=b+alpha\*t

print('Updated weights - w1 =',w1, 'w2 =', w2)

print('new bias',b)

AND\_Samples=[

[1,1,1],

[1,0,0],

[0,1,0],

[0,0,0] ]

OR\_Samples=[

[1,1,1],

[1,0,1],

[0,1,1],

[0,0,0] ]

XOR\_Samples=[

[1,1,0],

[1,0,1],

[0,1,1],

[0,0,0] ]

print('AND Sample')

delta\_rule(AND\_Samples)

print('OR Sample')

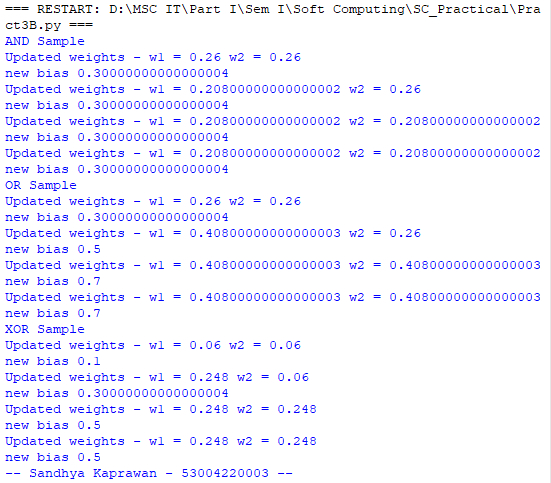
delta\_rule(OR\_Samples)

print('XOR Sample')

delta\_rule(XOR\_Samples)

print("-- Sandhya Kaprawan - 53004220003 --")

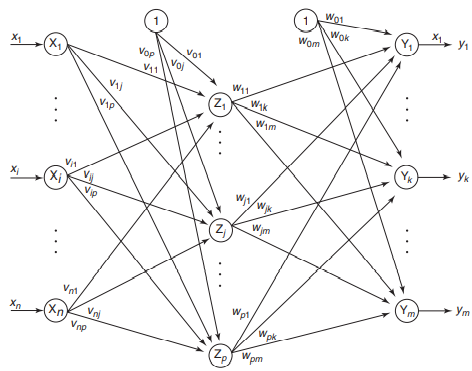
**Output:**



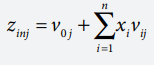
**Practical 4**

**A. Write a program for Forward Propagation Algorithm**

**Theory**:  In Forward propagation Neural Network, the input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer. In order to generate some output, the input data should be fed in the forward direction only. The data should not flow in reverse direction during output generation otherwise it would form a cycle and the output could never be generated. Such network configurations are known as feed-forward network. The feed-forward network helps in forward propagation. Architecture of Forward propagation algorithm is as follow:



In Hidden layer, we calculate weight and input signals and then calculate output of the hidden unit by applying its activation functions over z(inj)

Then y input will be calculated and activation function will be applied to computed output:

**Code:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

#scale units

X=X/np.amax(X,axis=0)

Y=Y/100;

class NN(object):

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

self.inputsize=2

self.outputsize=1

self.hiddensize=3

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize)

def forward(self,X):

self.z=np.dot(X,self.W1)

self.z2=self.sigmoidal(self.z)

self.z3=np.dot(self.z2,self.W2)

op=self.sigmoidal(self.z3)

return op;

def sigmoidal(self,s):

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

print("----- Forward Propgation -----")

obj=NN()

for i in range(5):

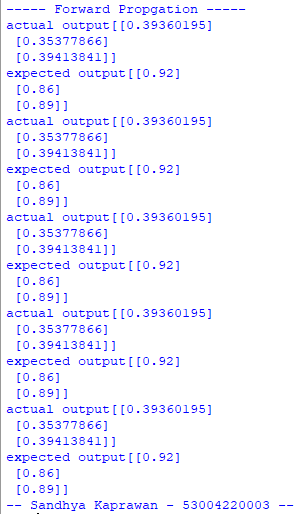
op=obj.forward(X)

print("actual output"+str(op))

print("expected output"+str(Y))

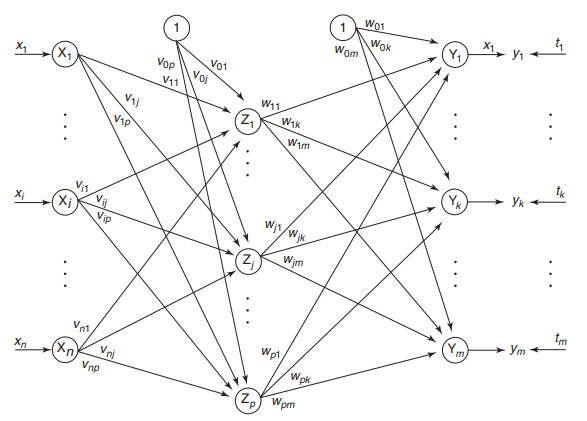
print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

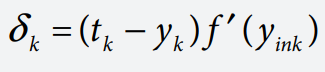


**B. Write a program for error Backpropagation algorithm**

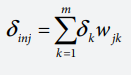
**Theory:** This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called back-propagation networks (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The back-propagation algorithm is different from other networks in respect to the process by which the weights are calculated during the learning period of the network. The general difficulty with the multilayer perceptron is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error. The training of the BPN is done in three stages – the feed-forward of the input training pattern, the calculation and back-propagation of the error, and updating of weights. The testing of the BPN involves the computation of feed-forward phase only. The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights.



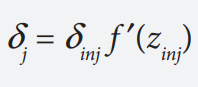
For Each output unit yk (k = 1 to m) we will calculate delta k and update weights and bias and send delta k backward to hidden layer

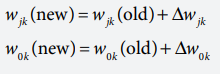
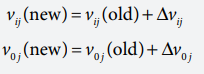
Each hidden unit (zj , j = 1 to p) sums its delta inputs from the output units



The term dinj gets multiplied with the derivative of f(zinj) to calculate the error term and on the basis of calculated error value update weight and bias

Now each hidden and output unit update the bias and weights accordingly

**Code:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

#scale units

X=X/np.amax(X,axis=0)

Y=Y/100;

class NN(object):

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

self.inputsize=2

self.outputsize=1

self.hiddensize=3

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize)

def forward(self,X):

self.z=np.dot(X,self.W1)

self.z2=self.sigmoidal(self.z)

self.z3=np.dot(self.z2,self.W2)

op=self.sigmoidal(self.z3)

return op;

def sigmoidal(self,s):

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

def sigmoidalprime(self,s):

return s\* (1-s)

def backward(self,X,Y,o):

self.o\_error=Y-o

self.o\_delta=self.o\_error \* self.sigmoidalprime(o)

self.z2\_error=self.o\_delta.dot(self.W2.T)

self.z2\_delta=self.z2\_error \* self.sigmoidalprime(self.z2)

self.W1 = self.W1 + X.T.dot(self.z2\_delta)

self.W2= self.W2+ self.z2.T.dot(self.o\_delta)

def train(self,X,Y):

o=self.forward(X)

self.backward(X,Y,o)

print("----- Back-propagation -----")

obj=NN()

for i in range(85):

print("----- Back-propagation -----")

print("input"+str(X))

print("Actual output"+str(Y))

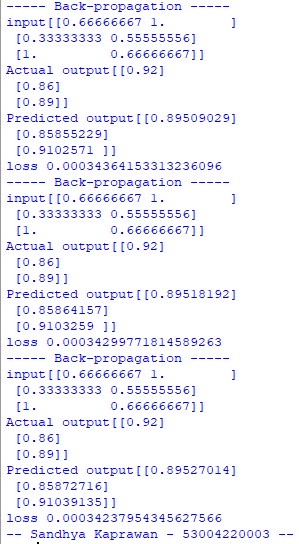
print("Predicted output"+str(obj.forward(X)))

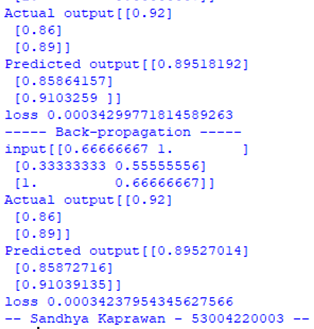
obj.train(X,Y)

print("loss "+str(np.mean(np.square(Y-obj.forward(X)))))

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

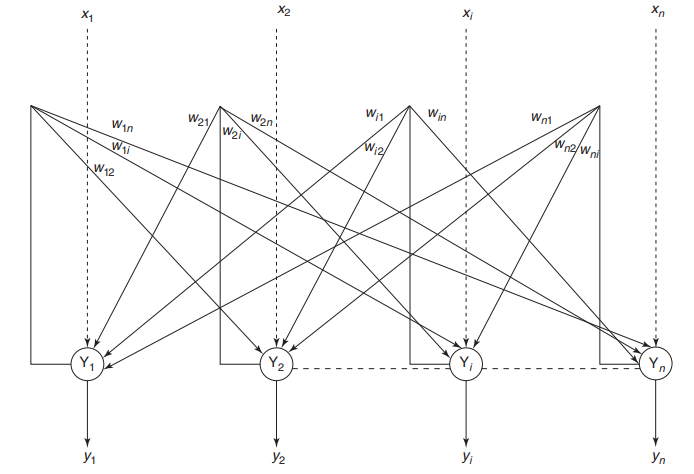




**Practical 5**

**A. Write a program for Hopfield Network**

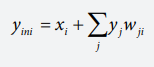
**Theory**:  John J. Hopfield developed a model in the year 1982 conforming to the asynchronous nature of biological neurons. The networks proposed by Hopfield are known as Hopfield networks and it is his work that promoted construction of the first analog VLSI neural chip. This network has found many useful applications in associative memory and various optimization problems. The asynchronous updation of the units allows a function, called as energy functions or Lyapunov function, for the net. The existence of this function enables us to prove that the net will converge to a stable set of activations. The usefulness of content addressable memory is realized by the discrete Hopfield net.



The network has symmetrical weights with no self-connections:



Calculate the net input of the network:



**Code:**

#Hopfield

import numpy as np

x=[[1,1,1,-1]]

w=[]

print(x)

print("Obtain weight matrix")

x1=np.transpose(x)

w=np.multiply(x,x1)

print(w)

print("Weight matrix with no self connection.")

for i in range(0,4):

for j in range(0,4):

if(i==j):

w[i][j]=0

print(w)

print("Converting to binary")

x\_bin=[]

y\_bin=[]

for i in range(0,4):

if(x[0][i]>=1):

x\_bin.append(1)

elif(x[0][i]<1):

x\_bin.append(0)

y\_bin=[x for x in x\_bin]

print(y\_bin)

print('Setting missing values')

x1\_bin=x\_bin

x1\_bin[0]=0

x1\_bin[1]=0

print('modified x1',x1\_bin)

y1\_bin=x1\_bin

print('modified y1',y1\_bin)

for i in range(0,4):

choice=int(input('Choose the y output: '))

var,op=0,0

if(choice==1):

for j in range(0,4):

print(x1\_bin[j],'\* ',w[j][0])

var=var+y1\_bin[j]\*w[j][0]

op=x1\_bin[0]+var

print('modified bit',op)

if(op>0):

op=1

elif(op<=0):

op=0

x1\_bin[0]=op

print(x1\_bin)

elif(choice==2):

for j in range(0,4):

var=var+y1\_bin[j]\*w[j][1]

op=x1\_bin[1]+var

if(op>0):

op=1

elif(op<=0):

op=0

x1\_bin[1]=op

print(x1\_bin)

elif(choice==3):

for j in range(0,4):

var=var+y1\_bin[j]\*w[j][2]

op=x1\_bin[2]+var

if(op>0):

op=1

elif(op<=0):

op=0

x1\_bin[2]=op

print(x1\_bin)

elif(choice==4):

for j in range(0,4):

var=var+y1\_bin[j]\*w[j][3]

op=x1\_bin[3]+var

if(op>0):

op=1

elif(op<=0):

op=0

x1\_bin[3]=op

print(x1\_bin)

#print('yyyyy',y\_bin)

if(x1\_bin==y\_bin):

print(x1\_bin,y\_bin)

print('No more iterations required')

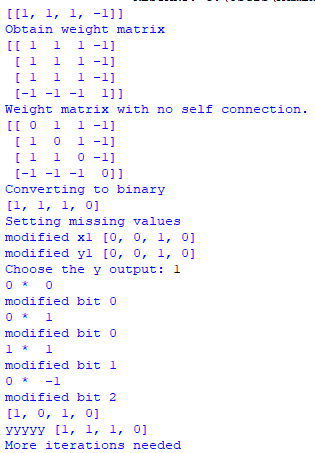
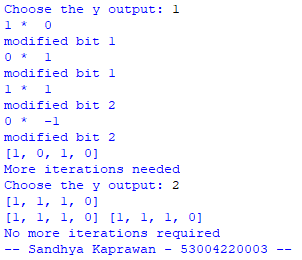
break;

else:

print('More iterations needed')

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

**Practical 6**

**A. Write a program for linear separation**

**Theory:** An ANN does not give an exact solution for a nonlinear problem. However, it provides possible approximate solutions to nonlinear problems. Linear separability is the concept wherein the separation of the input space into regions is based on whether the network response is positive or negative. A decision line is drawn to separate positive and negative responses. The decision line may also be called as the decision-making line or decision-support line or linear-separable line. The necessity of the linear separability concept was felt to classify the patterns based upon their output responses.

On the basis of the number of input units in the network, the above equation may represent a line, a plane or a hyperplane. The linear separability of the network is based on the decision-boundary line. If there exist weights (with bias) for which the training input vectors having positive (correct) response, +1, lie on one side of the decision boundary and all the other vectors having negative (incorrect) response, –1, lie on the other side of the decision boundary then we can conclude the problem is linearly separable. The separating line for which the boundary lies between the values x1 and x2 , so that the net gives a positive response on one side and negative response on other side

**Code:**

#Practical 7A - Linear separation.

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="g")

ax.scatter(0, 1, color="g")

ax.scatter(1, 0, color="g")

ax.scatter(1, 1, color="b")

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()

print("-- Sandhya Kaprawan - 53004220003 --")

#Distance Formula for linear 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-")

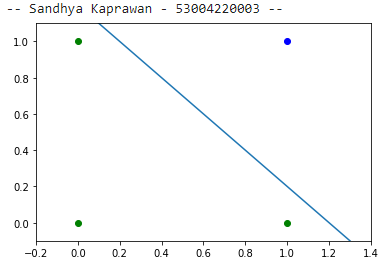
  else:

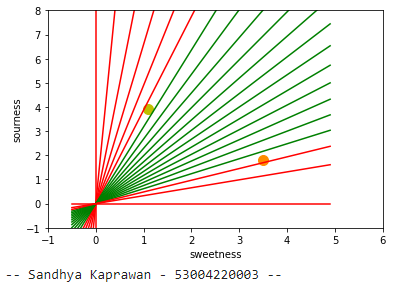
    ax.plot(X, Y, "r-")

plt.show()

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**





**Practical 7**

**A. Membership and Identity Operators | in, not in**

**Theory:** Python offers two membership operators to check or validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

in operator: The ‘in’ operator is used to check if a character/ substring/ element exists in a sequence or not. Evaluate to True if it finds the specified element in a sequence otherwise False. not in’ operator- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

**Code for in operator:**

#Practical 8A

list1=[1,2,3,4,5]

list2=[6,7,8,9]#[1,2,3,4,5]

for item in list1:

    if item in list2:

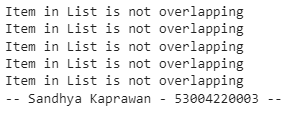
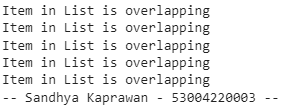
      print("Item in List is overlapping")

    else:

      print("Item in List is not overlapping")

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

**Code for not in operator:**

#not in operator

x = 24

y = 20

list = [10, 20, 30, 40, 50]

if (x not in list):

    print("x is NOT present in given list")

else:

    print("x is  present in given list")

if (y in list):

    print("y is present in given list")

else:

    print("y is NOT present in given list")

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

****

**B. Membership and Identity Operators is, is not**

**Theory:** Identity operators are used to compare the objects if both the objects are actually of the same data type and share the same memory location.  
There are different identity operators such as

1. **‘is’ operator –** Evaluates to True if the variables on either side of the operator point to the same object and false otherwise.

2. ‘is not’ operator - Returns true if both variables are not the same object

**Code:**

#Practical 8B

#dx = 5

x = 5

if (type(x) is int):

  print ("True:X is of type Integer")

else:

  print ("False:X is of type Integer")

y=10.0

if (type(y) is not int):

  print ("True:Y is not of type Integer")

else:

  print ("False:Y is of type Integer")

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**



**Practical 8**

**A. Find ratios using fuzzy logic**

**Theory:** FuzzyWuzzy is a library of Python which is used for string matching. Fuzzywuzzy uses a some similarity ratio between two sequences and returns the similarity percentage. We used the ratio() function to calculate the Levenshtein distance similarity ratio between the two strings (sequences).

The partial ratio() function allows us to perform substring matching.

The token\_sort\_ratio() function sorts the strings alphabetically and then joins them together. Then, the fuzz.ratio() is calculated.

The token\_set\_ratio() function is similar to the token\_sort\_ratio() function above, except it takes out the common tokens before calculating the fuzz.ratio() between the new strings.

FuzzyWuzzy also comes with a handy module, process, that returns the strings along with a similarity score out of a vector of strings.

**Code:**

#Prcatical 8A

#Installing FuzzyWuzzy

#!pip install fuzzywuzzy

#Import

import fuzzywuzzy

from fuzzywuzzy import fuzz

from fuzzywuzzy import process

print("Ratio finding using fuzzy logic")

Str\_A = 'FuzzyWuzzy is a lifesaver!'

Str\_B = 'fuzzy wuzzy is a LIFE SAVER.'

ratio = fuzz.ratio(Str\_A.lower(), Str\_B.lower())

print('Similarity score: {}'.format(ratio))

Str\_A = 'Chicago, Illinois'

Str\_B = 'Chicago'

ratio = fuzz.partial\_ratio(Str\_A.lower(), Str\_B.lower())

print('Similarity score: {}'.format(ratio))

Str\_A = 'Gunner William Kline'

Str\_B = 'Kline, Gunner William'

ratio = fuzz.token\_sort\_ratio(Str\_A, Str\_B)

print('Similarity score: {}'.format(ratio))

Str\_A = 'The 3000 meter steeplechase winner, Soufiane El Bakkali'

Str\_B = 'Soufiane El Bakkali'

ratio = fuzz.token\_set\_ratio(Str\_A, Str\_B)

print('Similarity score: {}'.format(ratio))

choices = ["Data Science", "Soft Computing","Research in Computing", "Cloud Computing"]

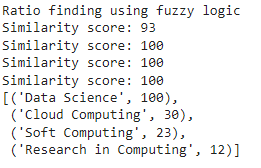
process.extract("Data Science", choices, scorer=fuzz.token\_sort\_ratio)

print("-- Sandhya Kaprawan - 53004220003 --")

choices = ["3000m Steeplechase", "Men's 3000 meter steeplechase","3000m STEEPLECHASE MENS", "mens 3000 meter SteepleChase"]

process.extractOne("Men's 3000 Meter Steeplechase", choices, scorer=fuzz.token\_sort\_ratio)

**Output:**

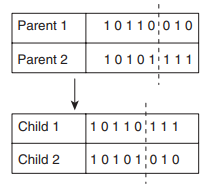
**Practical 9**

**A. Implementation of Simple genetic algorithm**

**Theory:**  Genetic Algorithms are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomized, GA’s are by no means random; instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, especially those that follow the principles first laid down by Charles Darwin, “survival of the fittest”, because in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

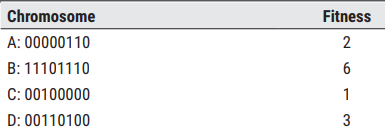
Genetic Algorithm is better than conventional algorithms in that they are more robust. Unlike older AI systems, they do not break easily even if the inputs are changed slightly or in the presence of reasonable noise. Also, in searching a large state-space, multimodal state-space or n-dimensional surface, GA offers significant benefits over more typical optimization techniques linear programming, heuristic, depth-first, breath-first and praxis, etc.

After selecting the parent chromosomes, GA performs the crossover. Crossover is the process of taking two parent solutions and producing from them a child. There are different types of crossover function available; one of them is single point crossover. The traditional genetic algorithm uses single-point crossover, where the two mating chromosomes are cut once at corresponding points and the sections after the cuts get exchanged. Here, a cross site or crossover point is selected randomly along the length of the mated strings and bits next to the cross sites are exchanged.



After crossover, the strings are subjected to mutation. Mutation prevents the algorithm to be trapped in a local minimum. Mutation plays the role of recovering the lost genetic materials as well as for randomly distributing genetic information. An important parameter in the mutation technique is the mutation probability (Pm). It decides how often parts of chromosome will be mutated. There are different types of mutation: Flipping, Interchanging, and Reversing.

Genetic Algorithm does the fitness test of offspring generated



**Code:**

import random

'''p1=[1,0,1,0,1,0]

p2=[1,1,0,0,1,1]

'''

p1=[]

p2=[]

p=[]

def fit(x):

print(x)

if (x.count(1)>=3):

return True

def crossover(p1,p2):

of1=p1[:2]+p2[2:]

of2=p2[0:2]+p1[2:]

print('offspring 1:\n',of1)

print('offspring 2:\n',of2)

if fit(mut(of1)):

p.append(of1)

if fit(mut(of2)):

p.append(of2)

def mut(x):

pos=random.choice(range(0,len(x)));

if x[pos]==1:

x[pos]=0

else:

x[pos]=1

return x

def select\_parent(p):

p1=random.choice(p)

p2=random.choice(p)

return [p1,p2]

print('--Practical 9 - Genetic Algorithm--')

print('\*\*Cromosome size should be greater than or equal to 3\*\*')

n=int(input("Enter the cromosome size: "))

print('Enter parent 1 chromosome')

for i in range(0,n):

p1.append(int(input('Enter P1: ')))

print('Enter parent 2 chromosome')

for i in range(0,n):

p2.append(int(input('Enter P2:')))

if fit(p1):

p.append(p1)

if fit(p2):

p.append(p2)

g1=int(input('No of generation to carry: '))

for i in range(0,g1):

select\_parent(p)

crossover(p1,p2)

print('New population:\n',p)

print('solutions:\n',p)

print("-- Sandhya Kaprawan - 53004220003 --")

**Output:**

