Keraleeya Samajam(Regd.) Dombivli's

MODEL COLLEGE

Re-Accredited Grade "A" by NAAC

Kanchan Goan Village, Khambalpada, Thakurli East – 421201
Contact No – 7045682157, 7045682158. www.model-college.edu.in

DEPARTMENT OF INFORMATION TECHNOLOGY AND COMPUTER SCIENCE

CERTIFICATE

This is to certify that Mr. /Miss ______________________________

*Studying in Class*______________*Seat No.* ________________

*Has completed the prescribed practicals in the subject*______________

*During the academic year*_________________________________

Date : ____________

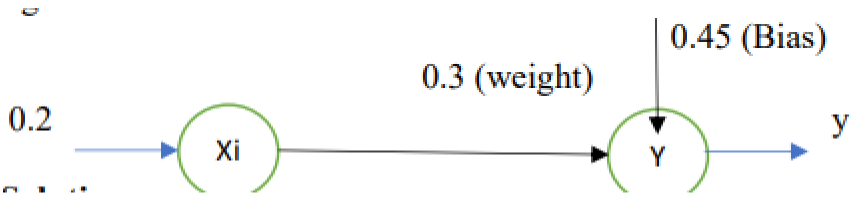
External Examiner

Internal Examiner
M.Sc. Information Technology

INDEX

SOFT COMPUTING TECHNIQUES PRACTICALS

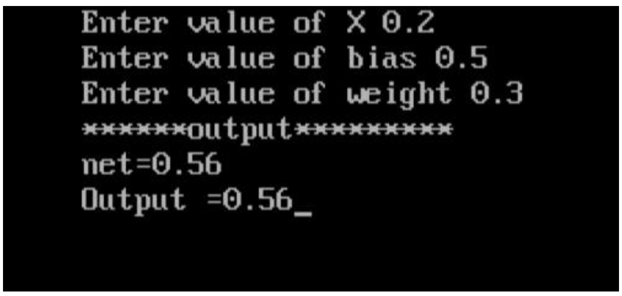
Practical No.01
Practical 1A: Design a simple linear neural network model
Problem: Create C++ program to calculate net input to the output neuron for the network shown in figure below.



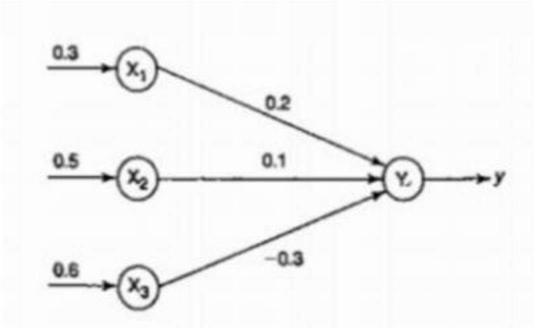
Code:

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
float x,b,w,net;
float out;
cout<<"Enter value of X";
cin>>x;
cout<<"Enter value of bias";
cin>>b;
cout<<"Enter value of weight";
cin>>w;
net=(w*x+b);
cout<<"******output*********";
cout<<"\nnet="<<net<<endl;
if(net<0)
{out=0;}
else if((net>=0)&&(net<=1))
{out=net;}
else
out=1;
cout<<"Output ="<<out;
getch();
}
```

OUTPUT:



Practical 1B: Calculate the output of neural net using both binary and bipolar sigmoidal function. For the network shown in the figure, calculate the net input to output neuron.



Code :

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
clrscr();
int i=0;
float x[10],b,w[10],net,n,sumxw=0,sigmo,e=2.71828;
float out;
cout<<"Enter the number of input : ";
cin>>n;
for (i=0;i<n;i++)
{
cout<<"Enter value of X"<<i+1;
cin>>x[i];
cout<<"Enter value of weight w"<<i+1;
cin>>w[i];
}
cout<<"Enter value of bias";
cin>>b;
for (i=0;i<n;i++)
{
sumxw=sumxw+w[i]*x[i];
}
net=(sumxw+b);
cout<<"******output*********";
cout<<"\nnet="<<net<<endl;
if(net<0)
{out=0;}
else if((net>=0)&&(net<=1))
{out=net;}
else
out=1;
cout<<"Output ="<<out;
cout<<"\n\n------------x------------";
cout<<"\n\nBinary sigmodial actication function : "<<(1/(1+(pow(e,-net))));
cout<<"\n\nBipolar sigmodial actication function : "<<(2/(1+(pow(e,-net))));
getch();
}
```

OUTPUT:

```
Enter the number of input : 3
Enter value of X1 2
Enter value of weight w10.2
Enter value of X2 3
Enter value of weight w20.2
Enter value of X3 4
Enter value of weight w30.2
Enter value of bias 0.5
******output*********
net=2.3
Output =1

------------x------------

Binary sigmodial actication function : 0.908877

Bipolar sigmodial actication function : 1.817754
```

PRACTICAL: 02

Practical 2A: Generate AND/NOT function using McCulloch-Pitts neural net.

CODE:

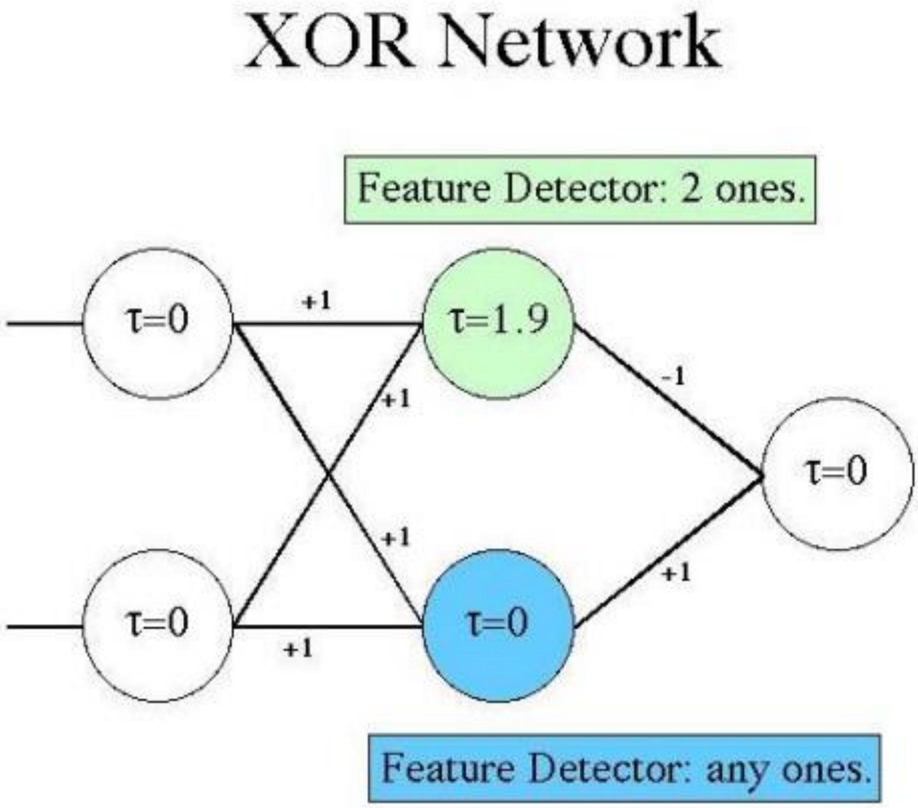
```
import numpy
num_ip=int(input("Enter the number of input: "))
w1 = 1
w2 = 1
print("For the",num_ip,"inpuets calculate the net inputs")
x1 = []
x2 = []
for j in range(0, num_ip):
        ele1 = int(input("x1 = "))
        ele2 = int(input("x2 = "))
        x1.append(ele1)
        x2.append(ele2)
print("x1 = ",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)
Yin = []
for i in range(0, num_ip):
        Yin. append(n[i] - m[i])
print("After assuming one weight as excitatory & other")
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:

```
2a.py - /Users/anjalinimje/Desktop/apython/2a.py (3.11.0)    IDLE Shell 3.11.0
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============================= RESTART: /Users/anjalinimje/Desktop/apython/2a.py =============================
Enter the number of input: 4
For the 4 inpuets calculate the net inputs
x1 = 0
x2 = 0
x1 = 0
x2 = 1
x1 = 1
x2 = 0
x1 = 1
x2 = 1
x1 =  [0, 0, 1, 1]
x2 =  [0, 1, 0, 1]
Yin =  [0, 1, 1, 2]
After assuming one weight as excitatory & other
Y =  [0, 0, 1, 0]
>>>
```

```
Enter the number of input: 4
For the 4 inpuets calculate the net inputs
x1 = 0
x2 = 0
x1 = 0
x2 = 1
x1 = 1
x2 = 0
x1 = 1
x2 = 1
x1 =  [0, 0, 1, 1]
x2 =  [0, 1, 0, 1]
Yin =  [0, 1, 1, 2]
After assuming one weight as excitatory & other
Y =  [0, 0, 1, 0]
```

Practical 2B: Generate XOR function using McCulloch-Pitts neural net.



Code:

```
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:

```
Neural Network Program
(1.0, 1.0, 0.014929208005738348, 0.0, 0.000222881251678602)
(0.0, 1.0, 0.9857295047367691, 1.0, 0.00020364703505789487)
(1.0, 0.0, 0.9856250336871464, 1.0, 0.00020663965649567642)
(0.0, 0.0, 0.016607849913409585, 0.0, 0.0002758206787463388)
[[ 0.          0.          5.75231929 -6.31595212  0.        ]
 [ 0.          0.         -5.97540997  6.18899346  0.        ]
 [ 0.          0.          0.          1.93019719  9.6814855 ]
 [ 0.          0.          0.          0.          9.57128428]
 [ 0.          0.          0.          0.          0.        ]]
[0.         0.         3.1933078  3.44466182 4.75885176]
```

Practical No: 03

Practical 3A: Write a program to implement Hebb's rule.

Code:

```
#include<iostream.h>
#include<conio.h>
void main()
{
float n,w,x=1,net,d,div,a,at=0.3,dw;
clrscr();
cout<<"Consider a single neuron perceptron with a single i/p";
cin>>w;
cout<<"\nEnter the learining coefficient";
cin>>d;
for(int i=0; i<10;i++)
{
net=x+w;
if(w<0)
a=0;
else
a=1;
div=at+a+w;
w=w+div;
cout<<"\ni+1 in fraction are i "<<a<<"\tchange in weight "<<div<<"\nadjustment at "<<w<< "\tnet
value is "<<net;
}
getch();
}
```

OUTPUT:

```
Consider a single neuron perceptron with a single i/p 1

Enter the learining coefficient 2

i+1 in fraction are i 1 change in weight2.3
adjustment at 3.3       net value is 2
i+1 in fraction are i 1 change in weight4.6
adjustment at 7.9       net value is 4.3
i+1 in fraction are i 1 change in weight9.2
adjustment at 17.099998 net value is 8.9
i+1 in fraction are i 1 change in weight18.399998
adjustment at 35.499996 net value is 18.099998
i+1 in fraction are i 1 change in weight36.799995
adjustment at 72.299988 net value is 36.499996
i+1 in fraction are i 1 change in weight73.599991
adjustment at 145.899979        net value is 73.299988
i+1 in fraction are i 1 change in weight147.199982
adjustment at 293.099976        net value is 146.899979
i+1 in fraction are i 1 change in weight294.399963
adjustment at 587.499939        net value is 294.099976
i+1 in fraction are i 1 change in weight588.799927
adjustment at 1176.299805       net value is 588.499939
i+1 in fraction are i 1 change in weight1177.599854
adjustment at 2353.899658       net value is 1177.299805
```

Python Code:

```
#Learning Rules #
import math
def computeNet(input, weights):
	net = 0
	for i in range(len(input)):
		net = net + input[i]*weights[i]
	print ("NET:")
	print (net)
	return net
#print ("NET:")
	#print net
	#return net
def computeFNetBinary(net):
	f_net = 0
	if(net>0):
		f_net = 1
	if(net<0):
		f_net = -1
	return f_net
def computeFNetCont(net):
	f_net = 0
	f_net = (2/(1+math.exp(-net)))-1
	return f_net
def hebb(f_net):
	return f_net
def perceptron(desired, actual):
	return (desired-actual)
def widrow(desired, actual):
	return (desired-actual)
def adjustWeights(inputs, weights, last, binary, desired, rule):
	c = 1
	if(last):
		print ("COMPLETE")
		return
	current_input = inputs[0]
	inputs = inputs[1:]
	if desired :
		current_desired = desired[0]
		desired = desired[1:]
	if len(inputs) == 0:
		last = True
	net = computeNet(current_input, weights)
	if(binary):
		f_net = computeFNetBinary(net)
	else:
		f_net = computeFNetCont(net)
	if rule == "hebb":
		r = hebb(f_net)
	elif rule == "perceptron":
		r = perceptron(current_desired, f_net)
	elif rule == "widrow":
		r = widrow(current_desired, net)
	del_weights = []
	for i in range(len(current_input)):
```

```
                x = (c*r)*current_input[i]
                del_weights.append(x)
                weights[i] = x
        print("NEW WEIGHTS:")
        print(weights)
        adjustWeights(inputs, weights, last, binary, desired, rule)
if __name__=="__main__":
        #total_inputs = (int)raw_input("Enter Total Number of Inputs)
        #vector_length = (int)raw_input("Enter Length of vector)
        total_inputs = 3
        vector_length = 4
        #for i in range(vector_length):
        #weight.append(raw_input("Enter Initial Weight:")
        weights = [1,-1,0,0.5]
        inputs = [[1,-2,1.5,0],[1,-0.5,-2,-1.5],[0,1,-1,1.5]]
        desired = [1,2,1,-1]
        print("BINARY HEBB!")
        adjustWeights(inputs, [1,-1,0,0.5], False, True, None, "hebb")
        print("CONTINUOUS HEBB!")
        adjustWeights(inputs, [1,-1,0,0.5], False, False, None, "hebb")
        print("PERCEPTRON!")
        adjustWeights(inputs, [1,-1,0,0.5], False, True, desired, "perceptron")
        print("WIDROW HOFF!")
        adjustWeights(inputs, [1,-1,0,0.5], False, True, desired, "widrow")
```

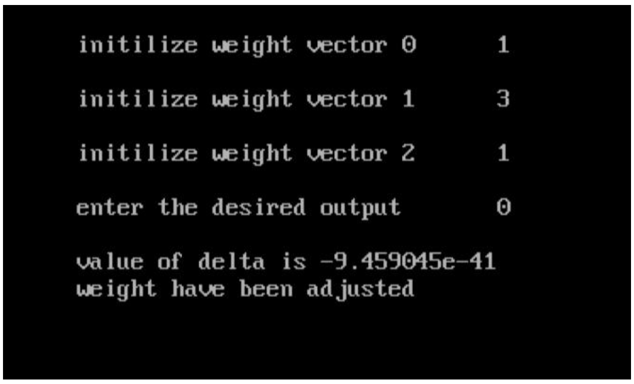
OUTPUT

```
BINARY HEBB!
NET:
3.0
NEW WEIGHTS:
[1, -2, 1.5, 0]
NET:
-1.0
NEW WEIGHTS:
[-1, 0.5, 2, 1.5]
NET:
0.75
NEW WEIGHTS:
[0, 1, -1, 1.5]
COMPLETE
CONTINUOUS HEBB!
NET:
3.0
NEW WEIGHTS:
[0.9051482536448667, -1.8102965072897335, 1.3577223804673002, 0.0]
NET:
-0.905148253644867
NEW WEIGHTS:
[-0.42401264054072996, 0.21200632027036498, 0.8480252810814599, 0.6360189608110949]
NET:
0.31800948040554744
NEW WEIGHTS:
[0.0, 0.15767814164392502, -0.15767814164392502, 0.23651721246588753]
COMPLETE
PERCEPTRON!
NET:
3.0
NEW WEIGHTS:
[0, 0, 0.0, 0]
NET:
0.0
NEW WEIGHTS:
[2, -1.0, -4, -3.0]
NET:
-1.5
NEW WEIGHTS:
[0, 2, -2, 3.0]
COMPLETE
WIDROW HOFF!
NET:
3.0
NEW WEIGHTS:
[-2.0, 4.0, -3.0, -0.0]
NET:
2.0
NEW WEIGHTS:
[0.0, -0.0, -0.0, -0.0]
NET:
0.0
NEW WEIGHTS:
[0.0, 1.0, -1.0, 1.5]
COMPLETE
```

3B practical: Write a program to implement of delta rule.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr( );
float input[3],d,del,a,val[10],w[10],weight[3],delta;
for(int i=0;i < 3 ; i++)
{
cout<<"\n initilize weight vector "<<i<<"\t";
cin>>input[i];
}
cout<<"\n enter the desired output\t";
cin>>d;
do
{
del=d-a;
if(del<0)
for(i=0 ;i<3 ;i++)
w[i]=w[i]-input[i];
else if(del>0)
for(i=0;i<3;i++)
weight[i]=weight[i]+input[i];
for(i=0;i<3;i++)
{
val[i]=del*input[i];
weight[+1]=weight[i]+val[i];
}
cout<<"\n value of delta is "<<del;
cout<<"\n weight have been adjusted";
}while(del==0);
if(del==0)
cout<<"\n output is correct";
getch();
}
```

OUTPUT:



Practical No:04

BACK PROPAGATION

4A: Write a program for Back Propagation Algorithm.

Python Code:

```
import math
import random
import sys

INPUT_NEURONS = 4
HIDDEN_NEURONS = 6
OUTPUT_NEURONS = 14

LEARN_RATE = 0.2 # Rho.
NOISE_FACTOR = 0.58
TRAINING_REPS = 10000
MAX_SAMPLES = 14

TRAINING_INPUTS = [[1, 1, 1, 0],
        [1, 1, 0, 0],
        [0, 1, 1, 0],
        [1, 0, 1, 0],
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [1, 1, 1, 1],
        [1, 1, 0, 1],
        [0, 1, 1, 1],
        [1, 0, 1, 1],
        [1, 0, 0, 1],
        [0, 1, 0, 1],
        [0, 0, 1, 1]]

TRAINING_OUTPUTS = [[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]

class Example_4x6x16:
        def __init__(self, numInputs, numHidden, numOutput, learningRate, noise, epochs,
         numSamples,
                inputArray, outputArray):
                self.mInputs = numInputs
                self.mHiddens = numHidden
                self.mOutputs = numOutput
```

```
        self.mLearningRate = learningRate
        self.mNoiseFactor = noise
        self.mEpochs = epochs
        self.mSamples = numSamples
        self.mInputArray = inputArray
        self.mOutputArray = outputArray
        self.wih = [] # Input to Hidden Weights
        self.who = [] # Hidden to Output Weights
        inputs = []
        hidden = []
        target = []
        actual = []
        erro = []
        errh = []
        return
    def initialize_arrays(self):
        for i in range(self.mInputs + 1): # The extra element represents bias node.
            self.wih.append([0.0] * self.mHiddens)
            for j in range(self.mHiddens):
                # Assign a random weight value between -0.5 and 0.5
                self.wih[i][j] = random.random() - 0.5
        for i in range(self.mHiddens + 1): # The extra element represents bias node.
            self.who.append([0.0] * self.mOutputs)
            for j in range(self.mOutputs):
                self.who[i][j] = random.random() - 0.5
        self.inputs = [0.0] * self.mInputs
        self.hidden = [0.0] * self.mHiddens
        self.target = [0.0] * self.mOutputs
        self.actual = [0.0] * self.mOutputs
        self.erro = [0.0] * self.mOutputs
        self.errh = [0.0] * self.mHiddens
        return
    def get_maximum(self, vector):
    # This function returns an array index of the maximum.
    index = 0
    maximum = vector[0]
    length = len(vector)

    for i in range(length):
        if vector[i] > maximum:
            maximum = vector[i]
            index = i
    return index

    def sigmoid(self, value):
        return 1.0 / (1.0 + math.exp(-value))
    def sigmoid_derivative(self, value):
        return value * (1.0 - value)
    def feed_forward(self):
        total = 0.0
        # Calculate input to hidden layer.
        for j in range(self.mHiddens):
            total = 0.0
            for i in range(self.mInputs):
                total += self.inputs[i] * self.wih[i][j]
```

```
                # Add in bias.
                total += self.wih[self.mInputs][j]
                self.hidden[j] = self.sigmoid(total)
        # Calculate the hidden to output layer.
        for j in range(self.mOutputs):
                total = 0.0
                for i in range(self.mHiddens):
                        total += self.hidden[i] * self.who[i][j]
                # Add in bias.
                total += self.who[self.mHiddens][j]
                self.actual[j] = self.sigmoid(total)
        return
    def back_propagate(self):
        # Calculate the output layer error (step 3 for output cell).
        for j in range(self.mOutputs):
                self.erro[j] = (self.target[j] - self.actual[j]) *
        self.sigmoid_derivative(self.actual[j])

        # Calculate the hidden layer error (step 3 for hidden cell).
        for i in range(self.mHiddens):
                self.errh[i] = 0.0
                for j in range(self.mOutputs):
                        self.errh[i] += self.erro[j] * self.who[i][j]
                self.errh[i] *= self.sigmoid_derivative(self.hidden[i])
        # Update the weights for the output layer (step 4).
        for j in range(self.mOutputs):
                for i in range(self.mHiddens):
                        self.who[i][j] += (self.mLearningRate * self.erro[j] * self.hidden[i])
        # Update the bias.
        self.who[self.mHiddens][j] += (self.mLearningRate * self.erro[j])
        # Update the weights for the hidden layer (step 4).
        for j in range(self.mHiddens):
                for i in range(self.mInputs):
                        self.wih[i][j] += (self.mLearningRate * self.errh[j] * self.inputs[i])
                # Update the bias.
                self.wih[self.mInputs][j] += (self.mLearningRate * self.errh[j])
        return
    def print_training_stats(self):
        sum = 0.0
        for i in range(self.mSamples):
                for j in range(self.mInputs):
                        self.inputs[j] = self.mInputArray[i][j]
                for j in range(self.mOutputs):
                        self.target[j] = self.mOutputArray[i][j]
                self.feed_forward()
                if self.get_maximum(self.actual) == self.get_maximum(self.target):
                        sum += 1
                else:
                        sys.stdout.write(str(self.inputs[0]) + "\t" + str(self.inputs[1]) + "\t" +
                str(self.inputs[2]) +
"\t" + str(self.inputs[3]) + "\n")
                        sys.stdout.write(str(self.get_maximum(self.actual)) + "\t" +
str(self.get_maximum(self.target)) + "\n")
        sys.stdout.write("Network is " + str((float(sum) / float(MAX_SAMPLES)) * 100.0) +
    "%
```

```
correct.\n")
			return
	def train_network(self):
			sample = 0
			for i in range(self.mEpochs):
					sample += 1
					if sample == self.mSamples:
							sample = 0
					for j in range(self.mInputs):
							self.inputs[j] = self.mInputArray[sample][j]
					for j in range(self.mOutputs):
							self.target[j] = self.mOutputArray[sample][j]
					self.feed_forward()
					self.back_propagate()
			return
	def test_network(self):
			for i in range(self.mSamples):
					for j in range(self.mInputs):
							self.inputs[j] = self.mInputArray[i][j]
					self.feed_forward()
					for j in range(self.mInputs):
							sys.stdout.write(str(self.inputs[j]) + "\t")
					sys.stdout.write("Output: " + str(self.get_maximum(self.actual)) + "\n")
			return
	def test_network_with_noise(self):
			# This function adds a random fractional value to all the training inputs greater than
	zero.
			for i in range(self.mSamples):
					for j in range(self.mInputs):
							self.inputs[j] = self.mInputArray[i][j] + (random.random() *
					NOISE_FACTOR)
					self.feed_forward()
					for j in range(self.mInputs):
							sys.stdout.write("{:03.3f}".format(((self.inputs[j] * 1000.0) /
					1000.0)) + "\t")
					sys.stdout.write("Output: " + str(self.get_maximum(self.actual)) + "\n")
			return
	if __name__ == '__main__':
			ex = Example_4x6x16(INPUT_NEURONS, HIDDEN_NEURONS,
	OUTPUT_NEURONS,
LEARN_RATE, NOISE_FACTOR, TRAINING_REPS, MAX_SAMPLES, TRAINING_INPUTS,
TRAINING_OUTPUTS)
	ex.initialize_arrays()
	ex.train_network()
	ex.print_training_stats()
	sys.stdout.write("\nTest network against original input:\n")
	ex.test_network()
	sys.stdout.write("\nTest network against noisy input:\n")
	ex.test_network_with_noise()
```

OUTPUT:

```
Network is 100.0% correct.

Test network against original input:
1       1       1       0       Output: 0
1       1       0       0       Output: 1
0       1       1       0       Output: 2
1       0       1       0       Output: 3
1       0       0       0       Output: 4
0       1       0       0       Output: 5
0       0       1       0       Output: 6
1       1       1       1       Output: 7
1       1       0       1       Output: 8
0       1       1       1       Output: 9
1       0       1       1       Output: 10
1       0       0       1       Output: 11
0       1       0       1       Output: 12
0       0       1       1       Output: 13

Test network against noisy input:
1.495   1.354   1.484   0.502   Output: 0
1.369   1.170   0.480   0.180   Output: 1
0.522   1.343   1.160   0.406   Output: 0
1.036   0.555   1.292   0.561   Output: 7
1.397   0.513   0.258   0.399   Output: 1
0.290   1.323   0.245   0.275   Output: 5
0.459   0.531   1.436   0.430   Output: 6
1.008   1.413   1.246   1.532   Output: 7
1.182   1.412   0.166   1.386   Output: 8
0.116   1.491   1.305   1.447   Output: 9
1.476   0.128   1.024   1.333   Output: 10
1.372   0.080   0.220   1.140   Output: 11
0.322   1.556   0.149   1.445   Output: 12
0.425   0.364   1.303   1.141   Output: 10
```

BACK PROPAGATION ERROR

Practical 4B: Write a program for error Backpropagation algorithm.

```
#include <conio>
#include<iostream>
 #include<math.h>
 void main()
 {
 clrscr();
 float l,c,s1,n1,n2,w10,b10,w20,b20,w11,b11,w21,b21,p,t,a0=-1,a1,a2,e,s2;
cout<<"enter the input weights/base of second n/w= ";
cin>>w10>>b10;
cout<<"enter the input weights/base of second n/w= ";
cin>>w20>>b20;
cout<<"enter the learning coefficient of n/w c= ";
cin>>c;
/* Step1:Propagation of signal through n/w */
n1=w10*p+b10;
a1=tanh(n1);
n2=w20*a1+b20;
a2=tanh(n2);
e=(t-a2); /* Back Propagation of Sensitivities */
s2=-2*(1-a2*a2)*e;
s1=(1-a1*a1)*w20*s2;
 /* Updation of weights and bases */
w21=w20-(c*s2*a1);
 w11=w10-(c*s1*a0);
b21=b20-(c*s2);
 b11=b10-(c*s1);
cout<<"The uploaded weight of first n/w w11= “<<w11;
cout"<<<"\n"<<"The uploaded weight of second n/w w21= “<<w21";
cout<<<"\n"<<"The uploaded base of second n/w b11= “<<b11;
cout"<<"\n"<<"The uploaded base of second n/w b21= "<<b21;
getch();
}
```

OUTPUT:

```
enter the input weights/base of second n/w= 0.25 -0.2
enter the input weights/base of second n/w= 0.45 0.3
enter the learning coefficient of n/w c= 23
The uploaded weight of first n/w w11= 4.21052
The uploaded weight of second n/w w21= 2.257548
The uploaded base of second n/w b11= -4.160521
The uploaded base of second n/w b21= -8.857923_
```

Python Code :

```
import math
import random
import sys
NUM_INPUTS = 3 # Input nodes, plus the bias input.
NUM_PATTERNS = 4 # Input patterns for XOR experiment.
NUM_HIDDEN = 4
NUM_EPOCHS = 200
LR_IH = 0.7 # Learning rate, input to hidden weights.
LR_HO = 0.07 # Learning rate, hidden to output weights.
# The data here is the XOR data which has been rescaled to the range -1 to 1.
# An extra input value of 1 is also added to act as the bias.
# e.g: [Value 1][Value 2][Bias]
TRAINING_INPUT = [[1, -1, 1], [-1, 1, 1], [1, 1, 1], [-1, -1, 1]]
# The output must lie in the range -1 to 1.
TRAINING_OUTPUT = [1, 1, -1, -1]

class Backpropagation1:
	def __init__(self, numInputs, numPatterns, numHidden, numEpochs, i2hLearningRate,
h2oLearningRate, inputValues, outputValues):
		self.mNumInputs = numInputs
		self.mNumPatterns = numPatterns
		self.mNumHidden = numHidden
		self.mNumEpochs = numEpochs
		self.mI2HLearningRate = i2hLearningRate
		self.mH2OLearningRate = h2oLearningRate
		self.hiddenVal = [] # Hidden node outputs.
		self.weightsIH = [] # Input to Hidden weights.
		self.weightsHO = [] # Hidden to Output weights.
		self.trainInputs = inputValues
		self.trainOutput = outputValues # "Actual" output values.
		self.errThisPat = 0.0
		self.outPred = 0.0 # "Expected" output values.
		self.RMSerror = 0.0 # Root Mean Squared error.
		return
	def initialize_arrays(self):
		# Initialize weights to random values.
		for j in range(self.mNumInputs):
			newRow = []
			for i in range(self.mNumHidden):
				self.weightsHO.append((random.random() - 0.5) / 2.0)
				weightValue = (random.random() - 0.5) / 5.0
				newRow.append(weightValue)
				sys.stdout.write("Weight = " + str(weightValue) + "\n")
			self.weightsIH.append(newRow)
			self.hiddenVal = [0.0] * self.mNumHidden
			return
		def calc_net(self, patNum):
			# Calculates values for Hidden and Output nodes.
			for i in range(self.mNumHidden):
				self.hiddenVal[i] = 0.0
				for j in range(self.mNumInputs):
					self.hiddenVal[i] += (self.trainInputs[patNum][j]
					self.weightsIH[j][i])
```

```
            self.hiddenVal[i] = math.tanh(self.hiddenVal[i])
        self.outPred = 0.0
        for i in range(self.mNumHidden):
            self.outPred += self.hiddenVal[i] * self.weightsHO[i]
        self.errThisPat = self.outPred - self.trainOutput[patNum] # Error =
        "Expected" - "Actual"
        return
    def adjust_hidden_to_output_weights(self):
        for i in range(self.mNumHidden):
            weightChange = self.mH2OLearningRate * self.errThisPat *
        self.hiddenVal[i]
            self.weightsHO[i] -= weightChange
            # Regularization of the output weights.
            if self.weightsHO[i] < -5.0:
                self.weightsHO[i] = -5.0
            elif self.weightsHO[i] > 5.0:
                self.weightsHO[i] = 5.0
        return
    def adjust_input_to_hidden_weights(self, patNum):
        for i in range(self.mNumHidden):
            for j in range(self.mNumInputs):
                x = 1 - math.pow(self.hiddenVal[i], 2)
                x = x * self.weightsHO[i] * self.errThisPat *
                self.mI2HLearningRate
                x = x * self.trainInputs[patNum][j]

                weightChange = x
                self.weightsIH[j][i] -= weightChange
        return
    def calculate_overall_error(self):
        errorValue = 0.0
    for i in range(self.mNumPatterns):
        self.calc_net(i)
        errorValue += math.pow(self.errThisPat, 2)
        errorValue /= self.mNumPatterns
        return math.sqrt(errorValue)
    def train_network(self):
        patNum = 0
        for j in range(self.mNumEpochs):
            for i in range(self.mNumPatterns):
                # Select a pattern at random.
                patNum = random.randrange(0, 4)

                # Calculate the output and error for this pattern.
                self.calc_net(patNum)

                # Adjust network weights.
                self.adjust_hidden_to_output_weights()
                self.adjust_input_to_hidden_weights(patNum)
            self.RMSerror = self.calculate_overall_error()
            sys.stdout.write("epoch = " + str(j) + " RMS Error = " +
        str(self.RMSerror) + "\n")
        return
    def display_results(self):
        for i in range(self.mNumPatterns):
```

```
                    self.calc_net(i)
                    sys.stdout.write("pat = " + str(i + 1) + " actual = " +
                str(self.trainOutput[i]) + " neural model
= " + str(self.outPred) + "\n")
                return
        if __name__ == '__main__':
                bp1 = Backpropagation1(NUM_INPUTS, NUM_PATTERNS,
        NUM_HIDDEN, NUM_EPOCHS,
LR_IH, LR_HO, TRAINING_INPUT, TRAINING_OUTPUT)
bp1.initialize_arrays()
bp1.train_network()
bp1.display_results()
```

OUTPUT:

```
Weight = 0.07434234733350246
Weight = 0.03847655245661426
Weight = -0.05008727568102127
Weight = 0.02202363798925413
Weight = 0.03442787627210346
Weight = 0.01856843595587565
Weight = 0.028318722749754267
Weight = 0.0925494546060405
Weight = 0.09362041215531455
Weight = -0.03403401942058752
Weight = 0.018564336622423537
Weight = 0.05659672781095375
epoch = 0 RMS Error = 1.0138158943263555
epoch = 1 RMS Error = 1.004145186163463
epoch = 2 RMS Error = 1.00913908094992
epoch = 3 RMS Error = 1.0191875412268172
epoch = 4 RMS Error = 1.006024685625373
epoch = 5 RMS Error = 1.0033477205227739
epoch = 6 RMS Error = 1.0007026759049533
epoch = 7 RMS Error = 1.0110829551692824
epoch = 8 RMS Error = 1.0256686813979496
epoch = 9 RMS Error = 1.005323911868788
epoch = 10 RMS Error = 1.000521830548554
epoch = 11 RMS Error = 0.9993682646525283
epoch = 12 RMS Error = 1.0038557853964178
epoch = 13 RMS Error = 0.9970727268388169
epoch = 14 RMS Error = 1.0266332959791744
epoch = 15 RMS Error = 0.9928860443017873
epoch = 16 RMS Error = 0.9964068085791118
```

epoch = 17 RMS Error = 1.0139348497023024
epoch = 18 RMS Error = 0.960299864909143
epoch = 19 RMS Error = 0.9525056298974087
epoch = 20 RMS Error = 0.9247126714170395
epoch = 21 RMS Error = 0.967096765312238
epoch = 22 RMS Error = 0.9096390947969907
epoch = 23 RMS Error = 0.9698936790623877
epoch = 24 RMS Error = 0.8826230561091781
epoch = 25 RMS Error = 0.9514852570209267
epoch = 26 RMS Error = 0.9596686811881783
epoch = 27 RMS Error = 1.0325632421537534
epoch = 28 RMS Error = 0.8275442669729415
epoch = 29 RMS Error = 0.8260202404175212
epoch = 30 RMS Error = 0.8350592948477509
epoch = 31 RMS Error = 0.8554352977515236
epoch = 32 RMS Error = 0.8297934806502157
epoch = 33 RMS Error = 0.833061930099073
epoch = 34 RMS Error = 0.8143052363856159
epoch = 35 RMS Error = 0.8176225206977333
epoch = 36 RMS Error = 0.8483452288109214
epoch = 37 RMS Error = 0.7595409671030884
epoch = 38 RMS Error = 0.7499416535186496
epoch = 39 RMS Error = 0.7756996239161544
epoch = 40 RMS Error = 0.731269911614097
epoch = 41 RMS Error = 0.7510344801939132
epoch = 42 RMS Error = 0.8896842535442822
epoch = 43 RMS Error = 0.9165883457794277
epoch = 44 RMS Error = 0.9559245773088916
epoch = 45 RMS Error = 0.819879777015132
epoch = 46 RMS Error = 0.8217289142664814
epoch = 47 RMS Error = 0.7386738874942209
epoch = 48 RMS Error = 0.7285492955760026
epoch = 49 RMS Error = 0.7224853942101742
epoch = 50 RMS Error = 0.7236313408843534
epoch = 51 RMS Error = 0.7900803524827611
epoch = 52 RMS Error = 0.7726552776437381
epoch = 53 RMS Error = 0.7205192305890895
epoch = 54 RMS Error = 0.7205204172829374
epoch = 55 RMS Error = 0.7864910289784377
epoch = 56 RMS Error = 0.9187213048129762
epoch = 57 RMS Error = 0.7942213030031665
epoch = 58 RMS Error = 0.8858091842838522
epoch = 59 RMS Error = 0.961497590253376
epoch = 60 RMS Error = 0.7716211309590792
epoch = 61 RMS Error = 0.7245657324680441
epoch = 62 RMS Error = 0.715452737737362
epoch = 63 RMS Error = 0.7415518927894849
epoch = 64 RMS Error = 0.8840178716451601
epoch = 65 RMS Error = 0.9394797833765706
epoch = 66 RMS Error = 0.7202145525668777
epoch = 67 RMS Error = 0.7172809465008269
epoch = 68 RMS Error = 0.7144922373039421
epoch = 69 RMS Error = 0.7133940924447338
epoch = 70 RMS Error = 0.7359165801829584
epoch = 71 RMS Error = 0.7324552152244883

epoch = 72 RMS Error = 0.8255356086321637
epoch = 73 RMS Error = 0.7246764025726955
epoch = 74 RMS Error = 0.8183763449668473
epoch = 75 RMS Error = 0.8126829983505828
epoch = 76 RMS Error = 0.8952260364004707
epoch = 77 RMS Error = 0.8070154257504034
epoch = 78 RMS Error = 0.7535500677311892
epoch = 79 RMS Error = 0.7340521036923598
epoch = 80 RMS Error = 0.8744931670014251
epoch = 81 RMS Error = 0.9058727833806055
epoch = 82 RMS Error = 0.7155691760735794
epoch = 83 RMS Error = 0.7167195727082317
epoch = 84 RMS Error = 0.7116955666919688
epoch = 85 RMS Error = 0.7147777343392939
epoch = 86 RMS Error = 0.7109204883356421
epoch = 87 RMS Error = 0.7134378563579238
epoch = 88 RMS Error = 0.7580566206561116
epoch = 89 RMS Error = 0.8549122357533895
epoch = 90 RMS Error = 0.8503524316110576
epoch = 91 RMS Error = 0.9198478207507859
epoch = 92 RMS Error = 0.9420432528360099
epoch = 93 RMS Error = 0.7607792875153779
epoch = 94 RMS Error = 0.7271376516233967
epoch = 95 RMS Error = 0.7113976232426222
epoch = 96 RMS Error = 0.7137561963314841
epoch = 97 RMS Error = 0.7249241564579542
epoch = 98 RMS Error = 0.7180834658564287
epoch = 99 RMS Error = 0.7669762508841187
epoch = 100 RMS Error = 0.7354740306090094
epoch = 101 RMS Error = 0.8042047908295417
epoch = 102 RMS Error = 0.7515716911966495
epoch = 103 RMS Error = 0.7137489720950695
epoch = 104 RMS Error = 0.8024993136892262
epoch = 105 RMS Error = 0.7144896762054247
epoch = 106 RMS Error = 0.7281662855387069
epoch = 107 RMS Error = 0.757784099534575
epoch = 108 RMS Error = 0.7435095006778665
epoch = 109 RMS Error = 0.7492705326725568
epoch = 110 RMS Error = 0.7100901645687169
epoch = 111 RMS Error = 0.7456986749120122
epoch = 112 RMS Error = 0.7305201196096227
epoch = 113 RMS Error = 0.8295251604187184
epoch = 114 RMS Error = 0.710247163485295
epoch = 115 RMS Error = 0.7119763812737366
epoch = 116 RMS Error = 0.7471487864079968
epoch = 117 RMS Error = 0.7424896059491721
epoch = 118 RMS Error = 0.7109356932105801
epoch = 119 RMS Error = 0.7430656945806488
epoch = 120 RMS Error = 0.7297257837801248
epoch = 121 RMS Error = 0.7095093610322196
epoch = 122 RMS Error = 0.7106818997499693
epoch = 123 RMS Error = 0.7768341432387924
epoch = 124 RMS Error = 0.7935583519932201
epoch = 125 RMS Error = 0.7928095276256659
epoch = 126 RMS Error = 0.7109328021136503

epoch = 127 RMS Error = 0.832019961794628
epoch = 128 RMS Error = 0.7190597479847527
epoch = 129 RMS Error = 0.7191138625170493
epoch = 130 RMS Error = 0.8637600738056124
epoch = 131 RMS Error = 0.892417880271405
epoch = 132 RMS Error = 0.7451922739681536
epoch = 133 RMS Error = 0.712487247387648
epoch = 134 RMS Error = 0.7148816975646508
epoch = 135 RMS Error = 0.7121074439485237
epoch = 136 RMS Error = 0.7314926755457392
epoch = 137 RMS Error = 0.7126172566733913
epoch = 138 RMS Error = 0.7509018014965921
epoch = 139 RMS Error = 0.8497321428298225
epoch = 140 RMS Error = 0.7334323992415763
epoch = 141 RMS Error = 0.7833763563989243
epoch = 142 RMS Error = 0.8152452526701938
epoch = 143 RMS Error = 0.7148833009035518
epoch = 144 RMS Error = 0.7542713108077818
epoch = 145 RMS Error = 0.7132426226394093
epoch = 146 RMS Error = 0.7089736296421195
epoch = 147 RMS Error = 0.7321999235557479
epoch = 148 RMS Error = 0.7785009735944584
epoch = 149 RMS Error = 0.709844405818833
epoch = 150 RMS Error = 0.7128325335652977
epoch = 151 RMS Error = 0.7125935814207374
epoch = 152 RMS Error = 0.8037721417613851
epoch = 153 RMS Error = 0.722687206636028
epoch = 154 RMS Error = 0.7227193371068649
epoch = 155 RMS Error = 0.7087967654327512
epoch = 156 RMS Error = 0.7480139218009648
epoch = 157 RMS Error = 0.7090826358997602
epoch = 158 RMS Error = 0.7407700622349834
epoch = 159 RMS Error = 0.7102485107544444
epoch = 160 RMS Error = 0.7768950905935967
epoch = 161 RMS Error = 0.7768144575488826
epoch = 162 RMS Error = 0.8268537664249791
epoch = 163 RMS Error = 0.7608283130284657
epoch = 164 RMS Error = 0.857113941742331
epoch = 165 RMS Error = 0.7716930647186001
epoch = 166 RMS Error = 0.8226524328860675
epoch = 167 RMS Error = 0.7164999311028765
epoch = 168 RMS Error = 0.8571237074737522
epoch = 169 RMS Error = 0.8932324636709013
epoch = 170 RMS Error = 0.8927400729912467
epoch = 171 RMS Error = 0.7351223026179372
epoch = 172 RMS Error = 0.7299767476669892
epoch = 173 RMS Error = 0.713986452466628
epoch = 174 RMS Error = 0.8055214045908644
epoch = 175 RMS Error = 0.8516766112438305
epoch = 176 RMS Error = 0.8891888366314729
epoch = 177 RMS Error = 0.7945274033467492
epoch = 178 RMS Error = 0.7316697441908177
epoch = 179 RMS Error = 0.8313292379503153
epoch = 180 RMS Error = 0.7776477540849469
epoch = 181 RMS Error = 0.7095912589903726

epoch = 182 RMS Error = 0.7123466598715167
epoch = 183 RMS Error = 0.7159073076441588
epoch = 184 RMS Error = 0.7195416216878733
epoch = 185 RMS Error = 0.7084389631143609
epoch = 186 RMS Error = 0.7860673843950462
epoch = 187 RMS Error = 0.8760905435188299
epoch = 188 RMS Error = 0.7480000089093307
epoch = 189 RMS Error = 0.712038860405997
epoch = 190 RMS Error = 0.8076620623839815
epoch = 191 RMS Error = 0.7132598047178094
epoch = 192 RMS Error = 0.7215591736319008
epoch = 193 RMS Error = 0.7196207802789324
epoch = 194 RMS Error = 0.7084566943460292
epoch = 195 RMS Error = 0.8383287401240865
epoch = 196 RMS Error = 0.9106847537746889
epoch = 197 RMS Error = 0.709991597625717
epoch = 198 RMS Error = 0.7296322834281765
epoch = 199 RMS Error = 0.7120836301943165
pat = 1 actual = 1 neural model= 0.10095163650022698
pat = 2 actual = 1 neural model= -0.1044556010479537
pat = 3 actual = -1 neural model= -0.9942305137817115
pat = 4 actual = -1 neural model= -0.989561418016595

Practical No:05

practical 5A: Write a program for Hopfield Network.

Given Pattern

[1,0,1,0] AND [0,1,0,1]

Given weighted vector

wt1 {0,-3,3,-3}

wt2{-3,0,-3,3}

wt3 {3,-3,0,-3}

wt4 {-3,3,-3,0}

Solution

Save HOP.H file in INCLUDE folder in C:\TurboC3\Include

HOP.H

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
class neuron
{
protected:
        int activation;
        friend class network;
public:
        int weightv[4];
        neuron() {};
        neuron(int *j) ;
        int act(int, int*);
};
        class network
{
public:
        neuron nrn[4];
        int output[4];
        int threshld(int) ;
        void activation(int j[4]);
        network(int*,int*,int*,int*);
};
```

________________________header file HOP.H ends here________________

Main program (hopnet.cpp)

```
#include "hop.h"
#include<conio.h>
neuron::neuron(int *j)
{
int i;
for(i=0;i<4;i++)
        {
        weightv[i]= *(j+i);
        }
}
int neuron::act(int m, int *x)
{
int i;
int a=0;
for(i=0;i<m;i++)
        {
        a += x[i]*weightv[i];
```

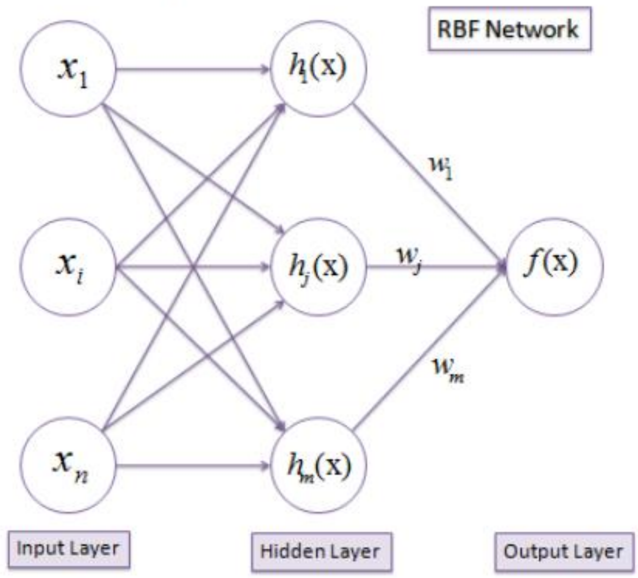
```
            }
return a;
}
int network::threshld(int k)
{
if(k>=0)
            return (1);
else
r           eturn (0);
}
network::network(int a[4],int b[4],int c[4],int d[4])
{
nrn[0] = neuron(a) ;
nrn[1] = neuron(b) ;
nrn[2] = neuron(c) ;
nrn[3] = neuron(d) ;
}
void network::activation(int *patrn)
{
int i,j;
for(i=0;i<4;i++)
            {
            for(j=0;j<4;j++)
                        {
                        cout<<"\n nrn["<<i<<"].weightv["<<j<<"] is "
                           <<nrn[i].weightv[j];
             }
nrn[i].activation = nrn[i].act(4,patrn);
cout<<"\nactivation is "<<nrn[i].activation;
output[i]=threshld(nrn[i].activation);
cout<<"\noutput value is "<<output[i]<<"\n";
            }
}
void main ()
{
int patrn1[]= {1,0,1,0},i;
int wt1[]= {0,-3,3,-3};
int wt2[]= {-3,0,-3,3};
int wt3[]= {3,-3,0,-3};
int wt4[]= {-3,3,-3,0};
cout<<"\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER
OF";
cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD
RECALL
THE";
cout<<"\nPATTERNS 1010 AND 0101 CORRECTLY.\n";
//create the network by calling its constructor.
// the constructor calls neuron constructor as many times as the number of
// neurons in the network.
network h1(wt1,wt2,wt3,wt4);
//present a pattern to the network and get the activations of the neurons
h1.activation(patrn1);
//check if the pattern given is correctly recalled and give message
for(i=0;i<4;i++)
            {
```

```
        if (h1.output[i] == patrn1[i])
                cout<<"\n pattern= "<<patrn1[i]<<
                " output = "<<h1.output[i]<<" component matches";
        else
                cout<<"\n pattern= "<<patrn1[i]<<
                " output = "<<h1.output[i]<<
                " discrepancy occurred";
        }
cout<<"\n\n";
int patrn2[]= {0,1,0,1};
h1.activation(patrn2);
for(i=0;i<4;i++)
        {
        if (h1.output[i] == patrn2[i])
                cout<<"\n pattern= "<<patrn2[i]<<
                " output = "<<h1.output[i]<<" component matches";
        else
                cout<<"\n pattern= "<<patrn2[i]<<
                " output = "<<h1.output[i]<<
                " discrepancy occurred";
        }
getch();
}
```

OUTPUT:

```
 nrn[1].weightv[0] is -3
 nrn[1].weightv[1] is 0
 nrn[1].weightv[2] is -3
 nrn[1].weightv[3] is 3
activation is 3
output value is 1

 nrn[2].weightv[0] is 3
 nrn[2].weightv[1] is -3
 nrn[2].weightv[2] is 0
 nrn[2].weightv[3] is -3
activation is -6
output value is 0

 nrn[3].weightv[0] is -3
 nrn[3].weightv[1] is 3
 nrn[3].weightv[2] is -3
 nrn[3].weightv[3] is 0
activation is 3
output value is 1

 pattern= 0 output = 0 component matches
 pattern= 1 output = 1 component matches
 pattern= 0 output = 0 component matches
 pattern= 1 output = 1 component matches_
```

5B. Write a program for Radial Basis function

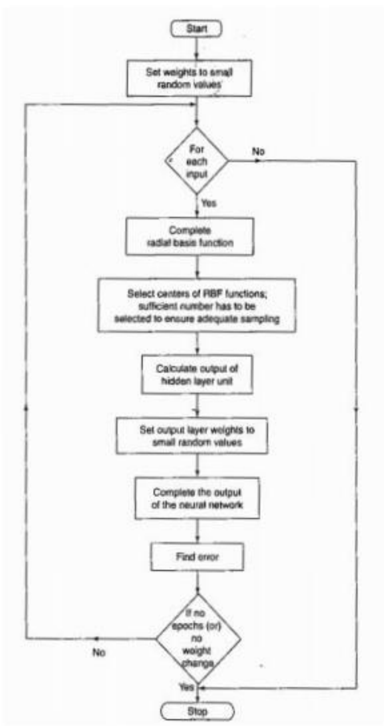


$$f(\mathrm{x}) = \sum_{j=1}^{m} w_j h_j(\mathrm{x})$$

$$h(x) = \exp\left(-\frac{(x-c)^2}{r^2}\right)$$

h(x) is the Gaussian activation function with the parameters r (the radius or standard deviation) and c (the center or average taken from the input space) defined separately at each RBF unit. The learning process is based on adjusting the parameters of the network to reproduce a set of input-output patterns. There are three types of parameters; the weight w between the hidden nodes and the output nodes, the center c of each neuron of the hidden layer and the unit width r.

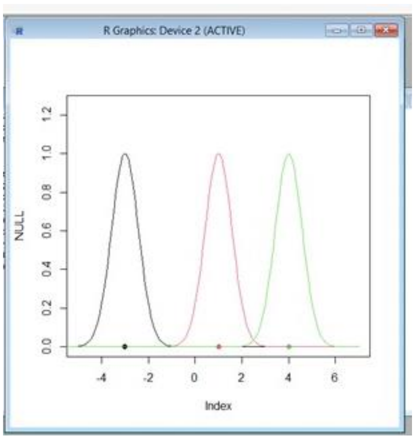
Algorithm



1.One-dimensional dataset as an illustration of the gaussian influence:

```
rbf.gauss <- function(gamma=1.0) {
        function(x) {
            exp(-gamma * norm(as.matrix(x),"F")^2 )
   }
}
D <- matrix(c(-3,1,4), ncol=1) # 3 datapoints
N <- length(D[,1])
xlim <- c(-5,7)
plot(NULL,xlim=xlim,ylim=c(0,1.25),type="n")
points(D,rep(0,length(D)),col=1:N,pch=19)
x.coord = seq(-7,7,length=250)
gamma <- 1.5
for (i in 1:N) {
        points(x.coord, lapply(x.coord - D[i,], rbf.gauss(gamma)), type="l", col=i)
}
```

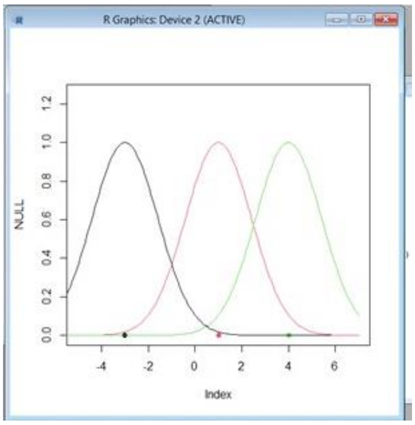
OUTPUT:



2.The value of gamma controls how far or how little the influnce of each datapoint is felt:

```
plot(NULL,xlim=xlim,ylim=c(0,1.25),type="n")
points(D,rep(0,length(D)),col=1:N,pch=19)
x.coord = seq(-7,7,length=250)
gamma <- 0.25
for (i in 1:N) {
	points(x.coord, lapply(x.coord - D[i,], rbf.gauss(gamma)), type="l", col=i)
}
```

OUTPUT:



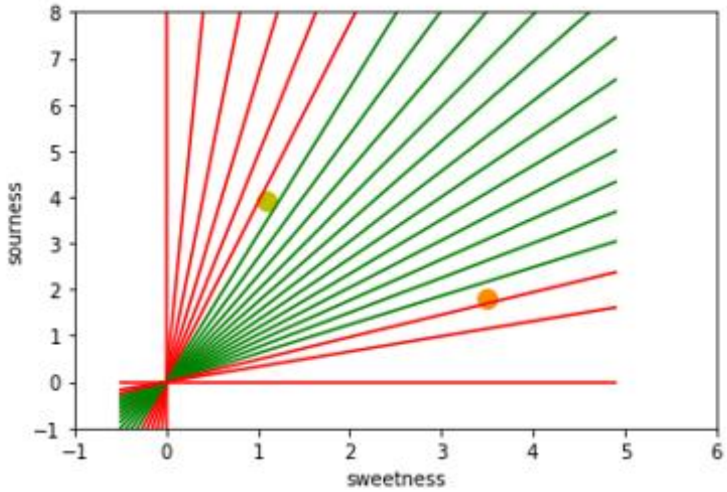
Practical No:06

6A Write a program for Linear separation.

Python code

```
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:07

7A: Membership and Identity Operators | in, not in
Python Code:

```
# Python program to illustrate
# Finding common member in list
# using 'in' operator
list1=[1,2,3,4,5]
list2=[6,7,8,9]
for item in list1:
	if item in list2:
		print("overlapping")
	else:
		print("not overlapping")
```

OUTPUT:

```
In [2]: list1=[1,2,3,4,5]
        list2=[6,7,8,9]
        for item in list1:
            if item in list2:
                print("overlapping")
        else:
            print("not overlapping")

        not overlapping
```

A.1. Membership and Identity Operators is, is not

```
# Python program to illustrate
# Finding common member in list
# without using 'in' operator
# Define a function() that takes two lists
def overlapping(list1,list2):
	c=0
	d=0
	for i in list1:
		c+=1
	for i in list2:
		d+=1
	for i in range(0,c):
		for j in range(0,d):
			if(list1[i]==list2[j]):
				return 1

	return 0
list1=[1,2,3,4,5]
list2=[6,7,8,9]
if (overlapping(list1, list2)):
	print("overlapping")
else:
	print ("not overlapping")
```

OUTPUT:

```
In [4]: def overlapping(list1,list2):
            c=0
            d=0
            for i in list1:
                c+=1
            for i in list2:
                d+=1
            for i in range(0,c):
                for j in range(0,d):
                    if(list1[i]==list2[j]):
                        return 1
            return 0
        list1=[1,2,3,4,5]
        list2=[6,7,8,9]
        if(overlapping(list1,list2)):
            print("overlapping")
        else:
            print("not overlapping")

        not overlapping
```

A.2 Python program to illustrate not 'in' operator

```
# Python program to illustrate
# 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")
```

OUTPUT:

```
In [5]: 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")

        x is NOT present in given list
        y is present in given list
```

7B

B.1. # Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
print ("true")
else:
print ("false")

OUTPUT:

```
In [6]: x = 5
        if (type(x) is int):
            print ("true")
        else:
            print ("false")

        true
```

B.2 # Python program to illustrate the
# use of 'is not' identity operator
x = 5.2
if (type(x) is not int):
print ("true")
else:
print ("false")

OUTPUT:

```
In [7]: x = 5.2
        if (type(x) is not int):
            print ("true")
        else:
            print ("false")

        true
```