

Course Code: ITITE26

Faculty Name: Mr. Satish Kr. Singh

Subject Name: ANN

**Submitted By**

**Aditya Gupta**

**2019UIT3112**

**Information Technology**

**INDEX**

|  |  |
| --- | --- |
| S.NO. | Program |
| 1 | Realize MCP for OR logic gate. |
| 2 | Realize MCP for XOR logic gate. |
| 3 | Implement Hebb learning rule for AND logic gate. |
| 4a | Implement Perceptron learning rule for AND NOT Logic gate. |
| 4b | Use Adaline networks to implement AND/OR Logic gate. |
| 4c | Implement backpropagation algorithm. |
| 4d | Implement Auto Associative for pattern association. |
| 4e | Implement Hetero Associative for pattern association. |
| 4f | Implement bidirectional associative memory for pattern association. |
| 4g | Implement Rail fence Cipher- Row and Column transformation |
| 5a | Implement Hopfield network for any given problem. |
| 5b | Implement Self organizing maps. |

## Q1. Realize MCP for OR logic gate.

#include<bits/stdc++.h>

using namespace std;

int main()

{

  int w1=1,w2=1;

  vector<vector<int>>inp={{0,0},{0,1},{1,0},{1,1}};

  cout<<"weights chosen :"<<endl;

  cout<<"w1=1"<<endl;

  cout<<"w2=1"<<endl;

  cout<<"Threshold chosen = 1"<<endl;

  cout<<"Inputs    yin    output"<<endl;

  for(int i=0;i<4;i++)

  {

    int yin=w1\*inp[i][0]+w2\*inp[i][1];

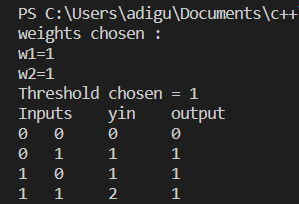
    int output=yin>=1?1:0;

    cout<<inp[i][0]<<"   "<<inp[i][1]<<"     "<<yin<<"      "<<output<<endl;

  }

}

**OUTPUT**

****

## Q2. Realize MCP for XOR logic gate.

#include<bits/stdc++.h>

using namespace std;

int main()

{

  cout<<"XOR is not linearly seperable, so we will be using two ANDNOT's and one OR MCP."<<endl;

  vector<vector<int>>inp={{0,0},{0,1},{1,0},{1,1}};

  cout<<"Weights chosen for 1st ANDNOT : 1 -1"<<endl;

  cout<<"Weights chosen for 2nd ANDNOT : -1 1"<<endl;

  cout<<"Threshold for both ANDNOT's : 1"<<endl;

  int w2=1,w3=1;

  cout<<"weights chosen for OR MCP: 1 1 "<<endl;

  cout<<"Threshold chosen for OR MCP: 1"<<endl;

  cout<<"Inputs    ANDNOT\_1    ANDNOT\_2    yin    output"<<endl;

  for(int i=0;i<4;i++)

  {

    int a=inp[i][0]-inp[i][1];

    a=a>=1?1:0;

    int b=inp[i][1]-inp[i][0];

    b=b>=1?1:0;

    int yin=w2\*a+w3\*b;

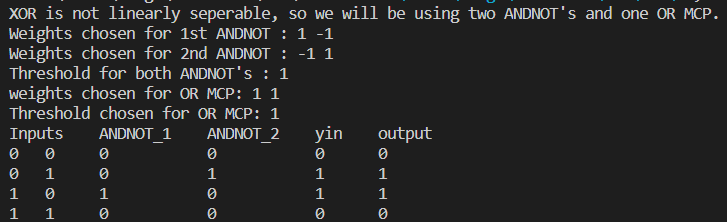
    int output=yin>=1?1:0;

    cout<<inp[i][0]<<"   "<<inp[i][1]<<"     "<<a<<"           "<<b<<"           "<<yin<<"      "<<output<<endl;

  }

}

**OUTPUT**



## Q3. Implement Hebb learning rule for AND logic gate.

#include<iostream>

#include<vector>

using namespace std;

class input{

public:

    int x1, x2;

    int correctOutput;

    int bias, w1, w2;

    input(int x1, int x2, int output){

        this->x1 = x1;

        this->x2 = x2;

        this->correctOutput = output;

        this->bias = 0;

        this->w1 = 0;

        this->w2 = 0;

    }

};

void hebbs(vector<input> in){

    int epoch=1;

    for(int i=0; i<in.size(); i++){

        cout<<"Input "<<i+1<<": \n";

        if(i!=0){

            in[i].w1 = in[i-1].w1;

            in[i].w2 = in[i-1].w2;

            in[i].bias = in[i-1].bias;

        }

        cout<<"Current weights: \n";

        cout<<"w1: "<<in[i].w1<<"    "<<"w2: "<<in[i].w2<<"    "<<"bias: "<<in[i].bias<<"\n";

        int newW1, newW2, newBias;

        newW1 = in[i].w1 + in[i].x1 \* in[i].correctOutput;

        newW2 = in[i].w2 + in[i].x2 \* in[i].correctOutput;

        newBias = in[i].bias + in[i].correctOutput;

        in[i].w1 = newW1;

        in[i].w2 = newW2;

        in[i].bias = newBias;

        cout<<"Updated weights: \n";

        cout<<"w1: "<<in[i].w1<<"    "<<"w2: "<<in[i].w2<<"    "<<"bias: "<<in[i].bias<<"\n\n";

        if(i == in.size()-1){

            in[0].w1 = in[i].w1;

            in[0].w2 = in[i].w2;

            in[0].bias = in[i].bias;

        }

    }

}

int main(){

    vector<input> in;

    int t;

    cout<<"Enter number of tuples"<<endl;

    cin>>t;

    cout<<"Give input for tuples in order of x1,x2,y"<<endl;

    while(t--){

        int tx1, tx2, tout;

        cin>>tx1>>tx2>>tout;

        input tin(tx1, tx2, tout);

        in.push\_back(tin);

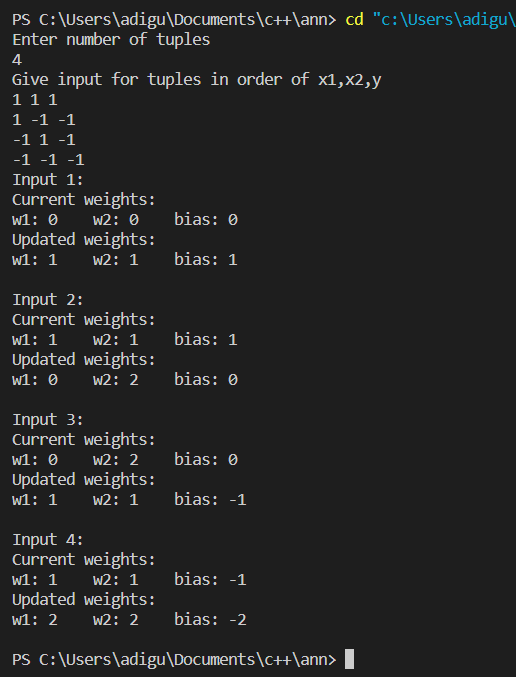
    }

    hebbs(in);

    return 0;

}

**OUTPUT:**

****

## Q4. Implement Perceptron learning rule for AND NOT Logic gate.

#include<iostream>

#include<vector>

using namespace std;

class input{

public:

   int x1, x2, correctOutput, w1, w2, bias;

   input(int x1, int x2, int correctOutput){

      this->x1 = x1;

      this->x2 = x2;

      this->correctOutput = correctOutput;

      this->w1 = 0;

      this->w2 = 0;

      this->bias = 0;

   }

};

class perceptron{

public:

   void model(vector<input> inputs, int learningRate){

      bool isChanged;

      int epochs=1;

      do{

         isChanged = false;

         cout<<"Epoch: "<<epochs++<<"\n";

         for(int i=0; i<4; i++){

            if(i!=0){

               inputs[i].w1 = inputs[i-1].w1;

               inputs[i].w2 = inputs[i-1].w2;

               inputs[i].bias = inputs[i-1].bias;

            }

            int yin = inputs[i].x1\*inputs[i].w1 + inputs[i].x2\*inputs[i].w2 + inputs[i].bias;

            if(yin>0)

               yin=1;

            else if(yin<0)

               yin=-1;

            else

               yin=0;

            if(yin != inputs[i].correctOutput){

               isChanged = true;

               inputs[i].w1 += learningRate\*inputs[i].correctOutput\*inputs[i].x1;

               inputs[i].w2 += learningRate\*inputs[i].correctOutput\*inputs[i].x2;

               inputs[i].bias += learningRate\*inputs[i].correctOutput;

            }

            if(i == 3){

               inputs[0].w1 = inputs[i].w1;

               inputs[0].w2 = inputs[i].w2;

               inputs[0].bias = inputs[i].bias;

            }

         }

         cout<<"w1: "<<inputs[0].w1<<" , w2: "<<inputs[0].w2<<" , bias: "<<inputs[0].bias<<"\n";

      }

      while(isChanged);

   }

};

int main(){

   vector<input> inputs;

   cout<<"Enter inputs in order x1,x2,y"<<endl;

   for(int i=0; i<4; i++){

      int x1, x2, y;

      cin>>x1>>x2>>y;

      inputs.push\_back(input(x1, x2, y));

   }

   int learningRate;

   cout<<"Enter learning rate"<<endl;

   cin>>learningRate;

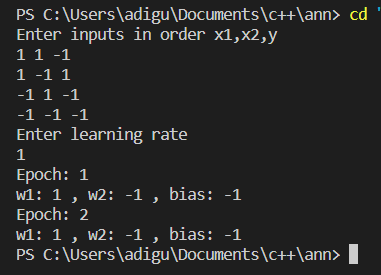
   perceptron p;

   p.model(inputs, learningRate);

   return 0;

}

**OUTPUT:**



**Q5. Use Adaline networks to implement AND/OR Logic gate.**

#include<iostream>

#include<ctime>

#include<cstdlib>

#include<vector>

using namespace std;

class input{

public:

   int x1, x2, y;

   input(int x1, int x2, int y){

      this->x1 = x1;

      this->x2 = x2;

      this->y = y;

   }

};

class Adaline{

public:

   void model(vector<input> inputs, float learningRate, int epochs, float w1, float w2, float bias){

      cout<<"Intial wts:\n";

      cout<<"w1: "<<w1<<" , w2: "<<w2<<" , bias: "<<bias<<"\n";

      int count=1;

      while(epochs--){

         cout<<"Epoch: "<<count++<<"\n";

         for(int i=0; i<inputs.size(); i++){

            float yin = inputs[i].x1\*w1 + inputs[i].x2\*w2 + bias;

            w1 += learningRate\*(inputs[i].y-yin)\*inputs[i].x1;

            w2 += learningRate\*(inputs[i].y-yin)\*inputs[i].x2;

            bias += learningRate\*(inputs[i].y-yin);

         }

         cout<<"w1: "<<w1<<" , w2: "<<w2<<" , bias: "<<bias<<"\n";

      }

      cout<<"Finished training";

   }

};

int main(){

   srand((unsigned)time(NULL));

   cout<<"Enter the no of epochs to run: ";

   int epochs;

   cin>>epochs;

   cout<<"Enter inputs in order x1,x2,y:\n";

   vector<input> inputs;

   int n=4;

   for(int i=0; i<n; i++){

      int x1, x2, y;

      cin>>x1>>x2>>y;

      inputs.push\_back(input(x1, x2, y));

   }

   cout<<"Enter learning rate: ";

   float learningRate;

   cin>>learningRate;

   cout<<"Enter initial values of w1, w2 & bias: ";

   float w1, w2, bias;

   cin>>w1>>w2>>bias;

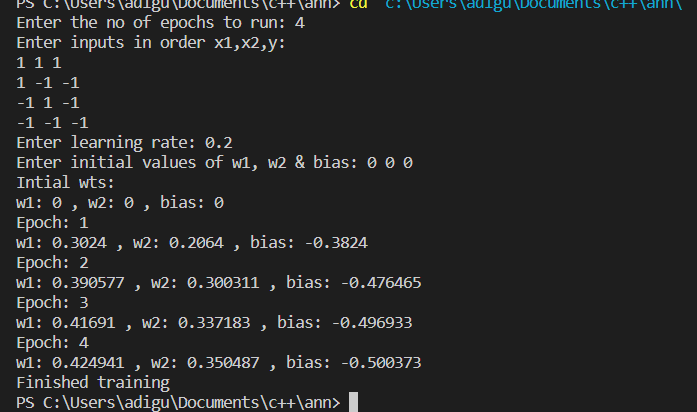
   Adaline model;

   model.model(inputs, learningRate, epochs, w1, w2, bias);

   return 0;

}

**OUTPUT:**

****

**Q6. Implement backpropagation algorithm.**

import numpy as np

import random

import math

class BPN:

    V = []

    W = []

    V0 = []

    W0 = []

    Zin = []

    Z = []

    Yin = []

    Y = []

    DelK = []

    DelW = []

    DelW0 = []

    DelinJ = []

    DelV = []

    DelV0 = []

    DelJ = []

    alpha = random.random()  # the learning rate

    def \_\_init\_\_(self, n, p, m):

        self.n = n

        self.p = p

        self.m = m

        # initialising the weights and biases

        self.V = self.DelV = np.array([ [ random.random() for \_ in range(self.p)] for \_ in range(self.n) ])

        self.W = self.DelW = np.array([ [ random.random() for \_ in range(self.m)] for \_ in range(self.p) ])

        self.V0 = self.DelV0 = np.array([ random.random() for \_ in range(self.p) ])

        self.W0 = self.DelW0 = np.array([ random.random() for \_ in range(self.m) ])

    def sigmoid(self,x):

        if type(x) is np.ndarray:

            return 1.0/(1.0 + np.exp(-x))

        else:

            return 1.0/(1.0 + math.exp(-x))

    def f(self, x):

        return self.sigmoid(x)

    def f\_dash(self, x):

        return self.f(x)\*(1 - self.f(x) )

    def feed\_forward(self, X):

        print( "---Feed Forward Phase---")

        ## This is the feed forward function, refer to the writeup above for more information

        print ("Weights(Input -> Hidden: ")

        print (self.V)

        print ("bias: ")

        print (self.V0)

        print ("Weights(Hidden -> Output): ")

        print (self.W)

        print ("bias: ")

        print (self.W0)

        for j in range(self.p):

            zinj = self.V0[j]

            for i in range(self.n):

                zinj+=X[i]\*self.V[i][j]

            (self.Zin).append(zinj)

        ## Now, we'll calculate Zj = f(Zinj)

        print ("Inputs(Hidden Layer): ")

        print (self.Zin)

        self.Z = self.f(np.array(self.Zin))

        print ("Outputs(Hidden Layer): ")

        print (self.Z)

        for k in range(self.m):

            yink = self.W0[k]

            for j in range(self.p):

                yink+=self.Z[j]\*self.W[j][k]

            (self.Yin).append(yink)

        print ("Net Input(Output Layer): ")

        print (self.Yin)

        self.Y = self.f(np.array(self.Yin))

        print ("Net Output(Output Layer): ")

        print (self.Y)

        print ("---End Feed Forward Phase---")

    def backpropagate(self, targets, X):

        ## Now, we'll begin with the backpropagation phase

        print ("---Back Propagation Phase---")

        ## Errors with each output layer neuron

        for k in range(self.m):

            delk = (targets[k] - self.Y[k])\*self.f\_dash(self.Yin[k])

            self.DelK.append(delk)

        print ("Output Errors for each Output Neuron: ")

        print (self.DelK)

        # Now we'll calculate the errors associated with the weights in Hidden and Output Layer

        for j in range(self.p):

            for k in range(self.m):

                self.DelW[j][k]  = self.alpha\*self.DelK[k]\*self.Z[j]

        # For Biases

        for k in range(self.m):

            self.DelW0[k] = self.alpha\*self.DelK[k]

        print ("Changes in weights (Hidden -> Output): ")

        print (self.DelW)

        print ("Change in bias: ")

        print (self.DelW0)

        ## Now,we'll calculate the backward error for each Hidden Layer Neuron

        for j in range(self.p):

            delj = sum( [ self.DelK[k]\*self.W[j][k] for k in range(self.m) ] )

            self.DelinJ.append(delj)

        print ("Input Errors with Each Hidden Layer Neuron: ")

        print (self.DelinJ)

        ## Output Errors for each Hidden Layer Neuron

        self.DelJ = np.array(self.DelinJ)\*self.f\_dash(np.array(self.Zin))

        print ("Output Errors for Each Hidden Layer Neuron: ")

        print (self.DelJ)

        ## Finally, we'll calculate the errors associated with weights joining Input and Hidden Layer

        for i in range(self.n):

            for j in range(self.p):

                self.DelV[i][j] = self.alpha\*self.DelJ[j]\*X[i]

        ## For Biases

        for j in range(self.p):

            self.DelV0[j] = self.alpha\*self.DelJ[j]

        print ("Changes in Weights (Input -> Hidden): ")

        print (self.DelV)

        print ("Change in bias: ")

        print (self.DelV0)

        print ("---End of Back Propagation Phase---")

    def update(self):

        print ("---Updation Phase Starts---")

        self.W  = self.W + self.DelW

        self.V  = self.V + self.DelV

        self.W0 = self.W0 + self.DelW0

        self.V0 = self.V0 + self.DelV0

        print ("Updated Weights(Input -> Hidden): ")

        print (self.V)

        print ("Updated bias: ")

        print (self.V0)

        print ("Updated Weights(Hidden -> Output): ")

        print (self.W)

        print ("Updated bias: ")

        print (self.W0)

        print ("---End of Updation Phase---")

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

    print ("Backpropagation Network: ")

    print ("parameters are randomly taken between 0 and 1")

    n,p,m = map(int, input("Enter number of neurons in Input, Hidden and Output Layers respectively (n,p,m):").split())

    epochs = int(input("Enter the number of Epochs:"))

    bpn = BPN(n,p,m)

    X = [ random.random() ]\*n           # the input vector

    targets = [ random.random() ]\*m     # the output vector

    for i in range(epochs):

        print ("\n---EPOCH", i+1, "---\n")

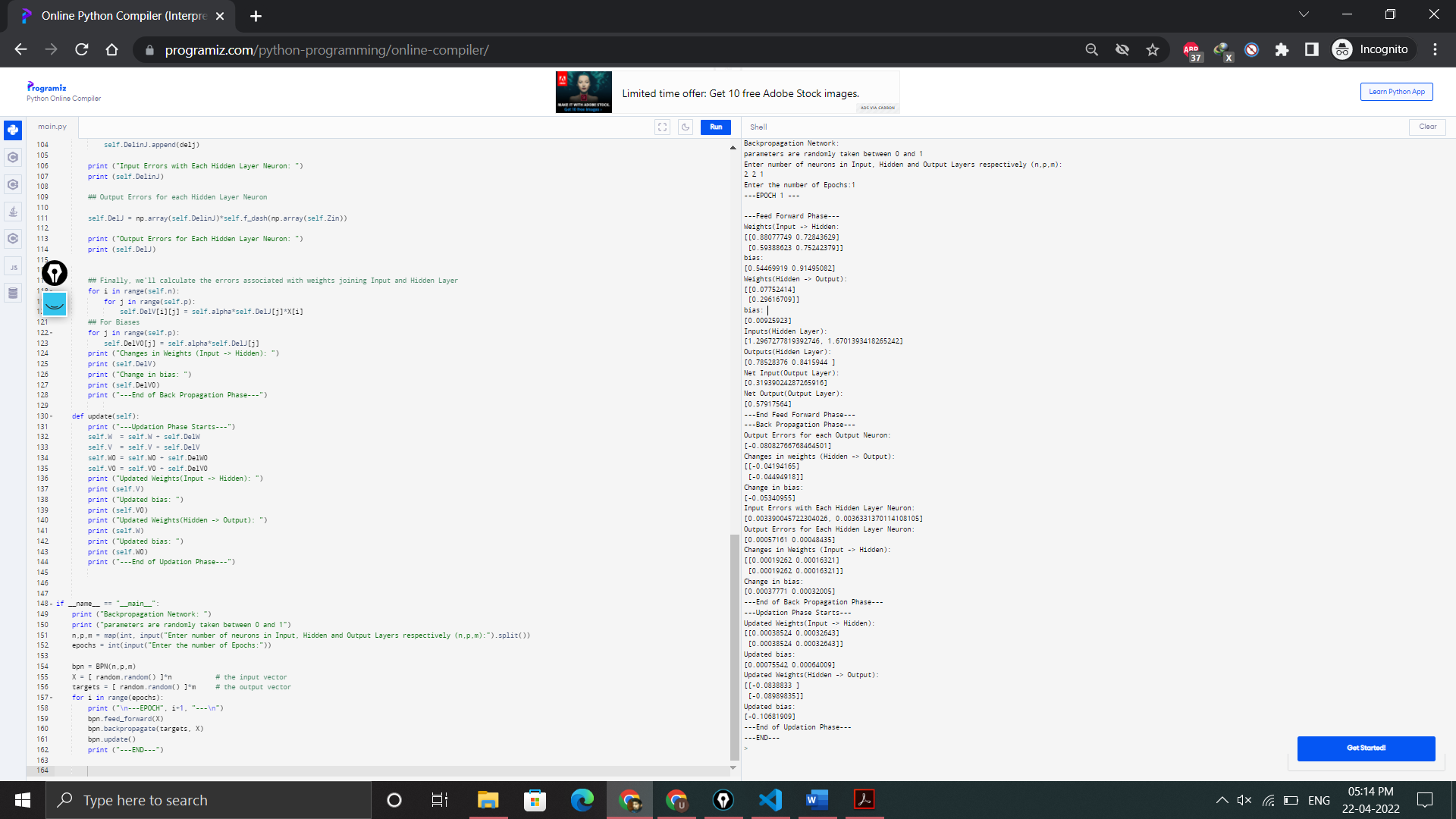
        bpn.feed\_forward(X)

        bpn.backpropagate(targets, X)

        bpn.update()

        print ("---END---")

**OUTPUT:**



**Q7-a. Implement Auto Associative for pattern association.**

#include<iostream>

#include<vector>

using namespace std;

class autoAssociative{

public:

   void model(vector<int> input, vector<int> test){

      vector<vector<int>> wt;

      for(int i=0; i<input.size(); i++){

         vector<int> tWt;

         for(int j=0; j<input.size(); j++){

            tWt.push\_back(input[i]\*input[j]);

         }

         wt.push\_back(tWt);

      }

      cout<<"Weight Matrix:\n";

      for(int i=0; i<wt.size(); i++){

         for(int j=0; j<wt[0].size(); j++)

            cout<<wt[i][j]<<" ";

         cout<<"\n";

      }

      cout<<"Test result: ";

      vector<int> result;

      for(int i=0; i<test.size(); i++){

         int temp=0;

         for(int j=0; j<test.size(); j++){

            temp += test[j]\*wt[j][i];

         }

         if(temp > 0)

            temp=1;

         else if(temp < 0)

            temp=-1;

         result.push\_back(temp);

         cout<<temp<<" ";

      }

   }

};

int main(){

   cout<<"Enter size of input vector: ";

   int n;

   cin>>n;

   cout<<"Enter the i/p vector: ";

   vector<int> input;

   for(int i=0; i<n; i++){

      int tin;

      cin>>tin;

      input.push\_back(tin);

   }

   cout<<"Enter the test vector: ";

   vector<int> test;

   for(int i=0; i<n; i++){

      int ttest; cin>>ttest;

      test.push\_back(ttest);

   }

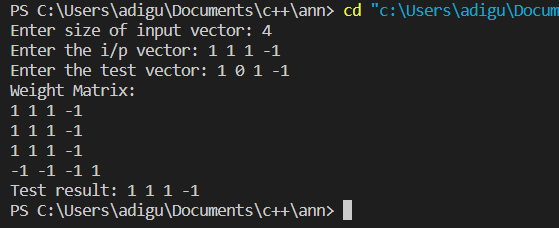
   autoAssociative model;

   model.model(input, test);

   return 0;

}

**OUTPUT:**



**Q7-b. Implement Hetero Associative for pattern association.**

#include<iostream>

#include<vector>

using namespace std;

class heteroAssociative{

public:

   void model(vector<vector<int>> input, vector<vector<int>> target, vector<int> test, int rowIn, int colIn, int rowTar, int colTar){

      vector<vector<int>> wt;

      for(int i=0; i<colIn; i++){

         vector<int> twt;

         for(int j=0; j<colTar; j++){

            int t=0;

            for(int k=0; k<rowIn; k++){

               t = t + input[k][i]\*target[k][j];

            }

            twt.push\_back(t);

         }

         wt.push\_back(twt);

      }

      cout<<"Weight Matrix:\n";

      for(int i=0; i<wt.size(); i++){

         for(int j=0; j<wt[0].size(); j++)

            cout<<wt[i][j]<<" ";

         cout<<"\n";

      }

      cout<<"Test result: ";

      vector<int> result;

      for(int j=0; j<wt[0].size(); j++){

         int temp=0;

         for(int i=0; i<test.size(); i++){

            temp += test[i]\*wt[i][j];

         }

         if(temp > 0)

            temp=1;

         else if(temp < 0)

            temp=-1;

         result.push\_back(temp);

         cout<<temp<<" ";

      }

   }

};

int main(){

   cout<<"Enter no of rows & columns of i/p vector: ";

   int rowIn, colIn;

   cin>>rowIn>>colIn;

   cout<<"Enter i/p matrix:\n";

   vector<vector<int>> input;

   for(int i=0; i<rowIn; i++){

      vector<int> tin;

      for(int j=0; j<colIn; j++){

         int t; cin>>t;

         tin.push\_back(t);

      }

      input.push\_back(tin);

   }

   cout<<"Enter no of rows & columns of target vector: ";

   int rowTar, colTar; cin>>rowTar>>colTar;

   cout<<"Enter target matrix:\n";

   vector<vector<int>> target;

   for(int i=0; i<rowTar; i++){

      vector<int> ttar;

      for(int j=0; j<colTar; j++){

         int t; cin>>t;

         ttar.push\_back(t);

      }

      target.push\_back(ttar);

   }

   cout<<"Enter test matrix:\n";

   vector<int> test;

   for(int i=0; i<colIn; i++){

      int t; cin>>t;

      test.push\_back(t);

   }

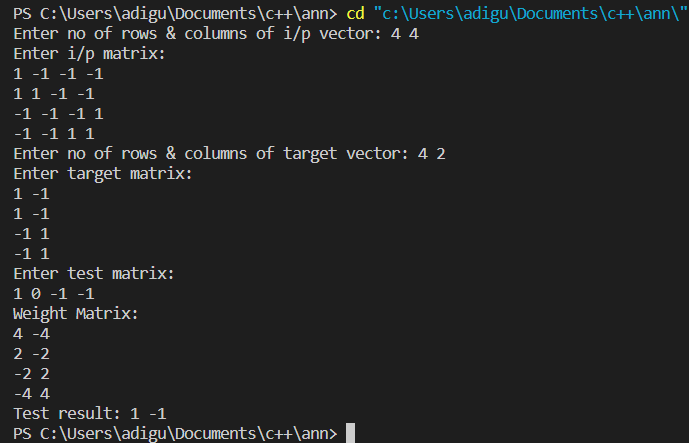
   heteroAssociative model;

   model.model(input, target, test, rowIn, colIn, rowTar, colTar);

   return 0;

}

**OUTPUT:**



**Q8. Implement bidirectional associative memory for pattern association.**

#include<iostream>

#include<vector>

using namespace std;

class BAM{

   vector<vector<int>> heteroassociative(vector<vector<int>> input, vector<vector<int>> target){

       int rowIn=input.size(), colIn=input[0].size(), rowTar=target.size(), colTar=target[0].size();

      vector<vector<int>> wt;

      for(int i=0; i<colIn; i++){

         vector<int> twt;

         for(int j=0; j<colTar; j++){

            int t=0;

            for(int k=0; k<rowIn; k++){

               t = t + input[k][i]\*target[k][j];

            }

            twt.push\_back(t);

         }

         wt.push\_back(twt);

      }

      return wt;

   }

public:

   void model(vector<vector<int>> input, vector<vector<int>> target, vector<int> testXY, vector<int> testYX){

      vector<vector<int>> wt = heteroassociative(input, target);

      cout<<"Weight Matrix for X->Y:\n";

      for(int i=0; i<wt.size(); i++){

         for(int j=0; j<wt[0].size(); j++)

            cout<<wt[i][j]<<" ";

         cout<<"\n";

      }

      cout<<"Test 1: X->Y :: [";

      for(int i=0; i<testXY.size(); i++)

         cout<<testXY[i]<<" ";

      cout<<"]\nTest result: ";

      vector<int> result;

      for(int j=0; j<wt[0].size(); j++){

         int temp=0;

         for(int i=0; i<testXY.size(); i++){

            temp += testXY[i]\*wt[i][j];

         }

         if(temp > 0)

            temp=1;

         else if(temp < 0)

            temp=-1;

         result.push\_back(temp);

         cout<<temp<<" ";

      }

      vector<vector<int>> wtT;

      for(int j=0; j<wt[0].size(); j++){

         vector<int> tWt;

         for(int i=0; i<wt.size(); i++){

            tWt.push\_back(wt[i][j]);

         }

         wtT.push\_back(tWt);

      }

      cout<<"Weight Matrix for Y->X:\n";

      for(int i=0; i<wtT.size(); i++){

         for(int j=0; j<wtT[0].size(); j++)

            cout<<wtT[i][j]<<" ";

         cout<<"\n";

      }

      cout<<"Test 2: Y->X :: [";

      for(int i=0; i<testYX.size(); i++)

         cout<<testYX[i]<<" ";

      cout<<"]\nTest result: ";

      result.clear();

      for(int j=0; j<wtT[0].size(); j++){

         int temp=0;

         for(int i=0; i<testYX.size(); i++){

            temp += testYX[i]\*wtT[i][j];

         }

         if(temp > 0)

            temp=1;

         else if(temp < 0)

            temp=-1;

         result.push\_back(temp);

         cout<<temp<<" ";

      }

   }

};

int main(){

   cout<<"Enter no of rows & columns of i/p vector: ";

   int rowIn, colIn;

   cin>>rowIn>>colIn;

   cout<<"Enter i/p matrix:\n";

   vector<vector<int>> input;

   for(int i=0; i<rowIn; i++){

      vector<int> tin;

      for(int j=0; j<colIn; j++){

         int t; cin>>t;

         tin.push\_back(t);

      }

      input.push\_back(tin);

   }

   cout<<"Enter no of rows & columns of target vector: ";

   int rowTar, colTar; cin>>rowTar>>colTar;

   cout<<"Enter target matrix:\n";

   vector<vector<int>> target;

   for(int i=0; i<rowTar; i++){

      vector<int> ttar;

      for(int j=0; j<colTar; j++){

         int t; cin>>t;

         ttar.push\_back(t);

      }

      target.push\_back(ttar);

   }

   cout<<"Enter test vectors:\n";

   vector<int> testXY;

   for(int i=0; i<colIn; i++){

      int t; cin>>t;

      testXY.push\_back(t);

   }

   vector<int> testYX;

   for(int i=0; i<colTar; i++){

      int t; cin>>t;

      testYX.push\_back(t);

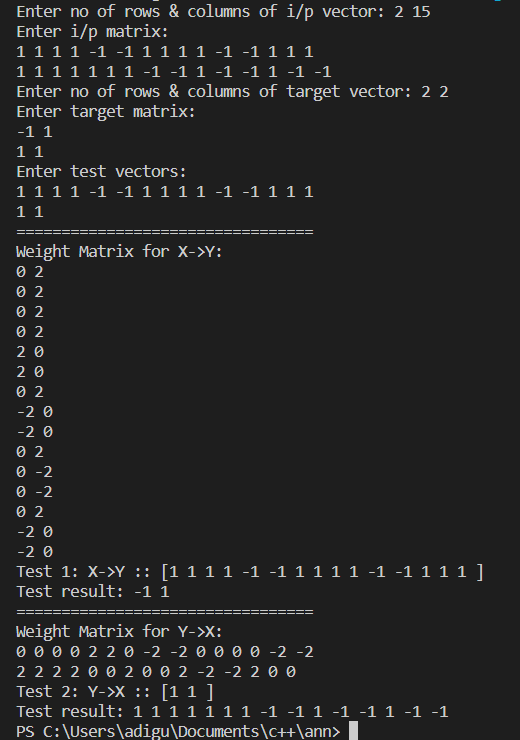
   }

   BAM model;

   model.model(input, target, testXY, testYX);

   return 0;

**OUTPUT**

****

**Q9. Implement Hopfield network for any given problem.**

#include<iostream>

#include<vector>

using namespace std;

class hopfield{

   vector<vector<int>> autoAssociative(vector<int> input){

      vector<vector<int>> wt;

      for(int i=0; i<input.size(); i++){

         vector<int> tWt;

         for(int j=0; j<input.size(); j++){

            if(i == j){

               tWt.push\_back(0);

               continue;

            }

            tWt.push\_back(input[i]\*input[j]);

         }

         wt.push\_back(tWt);

      }

      return wt;

   }

public:

   void model(vector<int> input, vector<int> test){

      vector<vector<int>> wt = autoAssociative(input);

      cout<<"Weight Matrix:\n";

      for(int i=0; i<wt.size(); i++){

         for(int j=0; j<wt[0].size(); j++)

            cout<<wt[i][j]<<" ";

         cout<<"\n";

      }

      for(int i=0; i<test.size(); i++)

         (test[i]<0)?test[i]=0:test[i];

      vector<int> y(test);

      for(int i=0; i<y.size(); i++){

         int prod=0;

         for(int j=0; j<y.size(); j++)

            prod += y[j]\*wt[j][i];

         y[i] = test[i] + prod;

         (y[i]<0)?y[i]=0:y[i];

         (y[i]>0)?y[i]=1:y[i];

      }

      cout<<"yin:\n";

      for(int i=0; i<y.size(); i++)

         cout<<y[i]<<" ";

      cout<<"\n";

      for(int i=0; i<input.size(); i++)

         (input[i]<0)?input[i]=0:input[i];

      for(int i=0; i<y.size(); i++){

         if(input[i] != y[i]){

            cout<<"Don't recognizes";

            return;

         }

      }

      cout<<"Recognizes";

   }

};

int main(){

   cout<<"Enter size of input vector: ";

   int n;

   cin>>n;

   cout<<"Enter the i/p vector: ";

   vector<int> input;

   for(int i=0; i<n; i++){

      int tin;

      cin>>tin;

      input.push\_back(tin);

   }

   cout<<"Enter the test vector: ";

   vector<int> test;

   for(int i=0; i<n; i++){

      int ttest; cin>>ttest;

      test.push\_back(ttest);

   }

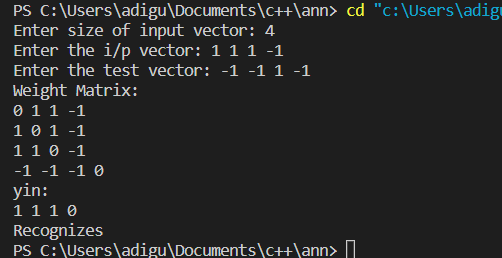
   hopfield model;

   model.model(input, test);

   return 0;

}

**OUTPUT:**



**Q10. Implement Self organizing maps.**

import math

class KohonenMap:

    def winner(self, weights, sample):

        D0 = 0

        D1 = 0

        for i in range(len(sample)):

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

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

            if D0 > D1:

                return 0

            else:

                return 1

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

        for i in range(len(weights)):

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

        return weights

def main():

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

    m, n = len(T), len(T[0])

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

    ob = KohonenMap()

    epochs = 3

    alpha = 0.5

    for i in range(epochs):

        for j in range(m):

            sample = T[j]

            J = ob.winner(weights, sample)

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

    s = [0, 0, 0, 1]

    J = ob.winner(weights, s)

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

    print("Trained weights: ", weights)

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

    main()

**OUTPUT:**

