Reg No:2117010
Program No:01
Date:03-01-2023
Program Name: Simulate McCullouch network for AND, OR and XOR gates

---

```
import numpy
from numpy import array
X = array([0,1,1])
W = array([-1,1,1])
Fx = W @ X
print(f'Input vector : {X} \nWeight Vector : {W}')
print(f'Weighted Sum : {Fx}')
#Threshold calculated based in the weighted sum
def linear_threshold_gate(Fx:int, T: float) -> int:
  '''Returns the binary threshold output'''
  if Fx >= T:
    return 1
  else:
    return 0

T = 1 # if threshold =1
activation = linear_threshold_gate(Fx,T)
print(f'Activation (IF threshold = 1): {activation}')

T = 3
activation = linear_threshold_gate(Fx,T)
print(f'Activation (IF threshold = 3): {activation}')
```

OUTPUT:

```
Input vector : [0 1 1]
Weight Vector : [-1  1  1]
Weighted Sum : 2
```

```
Activation (IF threshold = 1): 1
Activation (IF threshold = 3): 0
```

Reg No:2117010
Program NO:02
Date:03-01-2023
Program Name: Simulate Hebb learning network for AND, OR and XOR gates

---

```
def habbian_learning(samples):
    w1, w2, b = 0, 0 ,0
    for x1, x2, y in samples:
        w1 = w1 + x1 * y
        w2 = w2 + x2 * y
        b = b + y
        print(f'({x1:2}, {x2:2}) {y:2} ({x1:2}, {x2:2}, {y:2}) ({w1:2}, {w2:2}, {b:2})')
AND_samples = {
    'binary_input_binary_output' : [
        [1,1,1],
        [1,0,0],
        [0,1,0],
        [0,0,0]
    ],
    'binary_input_bipolar_output' : [
        [1,1,1],
        [1,0,-1],
        [0,1,-1],
        [0,0,-1]
    ],
    'bipolar_input_bipolar_output' : [
        [1,1,1],
        [1,-1,-1],
        [-1,1,-1],
        [-1,-1,-1]
    ]
}
OR_samples = {
    'binary_input_binary_output':[
        [1,1,1],
        [1,0,1],
        [0,1,1],
        [0,0,0]
    ],
    'binary_input_bipolar_output' :[
        [1,1,1],
        [1,0,1],
        [0,1,1],
        [0,0,-1]
    ],
    'bipolar_input_bipolar_output' :[
        [1,1,1],
        [1,-1,1],
        [-1,1,1],
        [-1,-1,-1]
    ]}
XOR_samples = {
    'binary_input_binary_output' :[
```

```
        [1,1,0],
        [1,0,1],
        [0,1,1],
        [0,0,0]
    ],
    'binary_input_bipolar_output' :[
        [1,1,-1],
        [1,0,1],
        [0,1,1],
        [0,0,-1]
    ],
    'bipolar_input_bipolar_output' :[
        [1,1,-1],
        [1,-1,1],
        [-1,1,1],
        [-1,-1,-1]
    ]
}
print('\n', 'Hebbian Learning', '\n')
print("AND_SAMPLES")
print('AND binary_input_binary_output ')
habbian_learning(AND_samples['binary_input_binary_output'])
print('AND binary_input_bipolar_output ')
habbian_learning(AND_samples['binary_input_bipolar_output'])
print('AND bipolar_input_bipolar_output')
habbian_learning(AND_samples['bipolar_input_bipolar_output'])

print("OR_SAMPLES")
print('OR binary_input_binary_output ')
habbian_learning(OR_samples['binary_input_binary_output'])
print('OR binary_input_bipolar_output ')
habbian_learning(OR_samples['binary_input_bipolar_output'])
print('OR bipolar_input_bipolar_output')
habbian_learning(OR_samples['bipolar_input_bipolar_output'])

print("XOR_SAMPLES")
print('XOR binary_input_binary_output ')
habbian_learning(XOR_samples['binary_input_binary_output'])
print('XOR binary_input_bipolar_output ')
habbian_learning(XOR_samples['binary_input_bipolar_output'])
print('XOR bipolar_input_bipolar_output')
habbian_learning(XOR_samples['bipolar_input_bipolar_output'])
```

OUTPUT:

```
 Hebbian Learning

AND_SAMPLES
AND binary_input_binary_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0)  0 ( 1,  0,  0) ( 1,  1,  1)
( 0,  1)  0 ( 0,  1,  0) ( 1,  1,  1)
( 0,  0)  0 ( 0,  0,  0) ( 1,  1,  1)
AND binary_input_bipolar_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0) -1 ( 1,  0, -1) ( 0,  1,  0)
( 0,  1) -1 ( 0,  1, -1) ( 0,  0, -1)
( 0,  0) -1 ( 0,  0, -1) ( 0,  0, -2)
AND bipolar_input_bipolar_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1, -1) -1 ( 1, -1, -1) ( 0,  2,  0)
(-1,  1) -1 (-1,  1, -1) ( 1,  1, -1)
(-1, -1) -1 (-1, -1, -1) ( 2,  2, -2)
```

```
OR_SAMPLES
OR binary_input_binary_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0)  1 ( 1,  0,  1) ( 2,  1,  2)
( 0,  1)  1 ( 0,  1,  1) ( 2,  2,  3)
( 0,  0)  0 ( 0,  0,  0) ( 2,  2,  3)
OR binary_input_bipolar_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1,  0)  1 ( 1,  0,  1) ( 2,  1,  2)
( 0,  1)  1 ( 0,  1,  1) ( 2,  2,  3)
( 0,  0) -1 ( 0,  0, -1) ( 2,  2,  2)
OR bipolar_input_bipolar_output
( 1,  1)  1 ( 1,  1,  1) ( 1,  1,  1)
( 1, -1)  1 ( 1, -1,  1) ( 2,  0,  2)
(-1,  1)  1 (-1,  1,  1) ( 1,  1,  3)
(-1, -1) -1 (-1, -1, -1) ( 2,  2,  2)

XOR_SAMPLES
XOR binary_input_binary_output
( 1,  1)  0 ( 1,  1,  0) ( 0,  0,  0)
( 1,  0)  1 ( 1,  0,  1) ( 1,  0,  1)
( 0,  1)  1 ( 0,  1,  1) ( 1,  1,  2)
( 0,  0)  0 ( 0,  0,  0) ( 1,  1,  2)
XOR binary_input_bipolar_output
( 1,  1) -1 ( 1,  1, -1) (-1, -1, -1)
( 1,  0)  1 ( 1,  0,  1) ( 0, -1,  0)
( 0,  1)  1 ( 0,  1,  1) ( 0,  0,  1)
( 0,  0) -1 ( 0,  0, -1) ( 0,  0,  0)
XOR bipolar_input_bipolar_output
( 1,  1) -1 ( 1,  1, -1) (-1, -1, -1)
( 1, -1)  1 ( 1, -1,  1) ( 0, -2,  0)
(-1,  1)  1 (-1,  1,  1) (-1, -1,  1)
(-1, -1) -1 (-1, -1, -1) ( 0,  0,  0)
```

Reg No:2117010
Program No:03
Date:03-01-2023
Program Name: Simulate Perceptron network for AND, OR and XOR gates

---

```
import numpy as np
def unitStep(v):
    if v >= 0:
        return 1
    else:
        return 0

def perceptronModel(x,w,b):
    v = np.dot(w,x)+b
    y = unitStep(v)
    return y

def NOT_logic(x):
    wNOT = -1
    bNOT = 0.5
    return perceptronModel(x, wNOT, bNOT)

def AND_logic(x):
    w = np.array([1,1])
    bAND = -1.5
    return perceptronModel(x, w, bAND)

def OR_logic(x):
    w = np.array([1,1])
    bOR = -0.5
    return perceptronModel(x, w, bOR)

def XOR_logic(x):
    y1 = AND_logic(x)
    y2 = OR_logic(x)
    y3 = NOT_logic(y1)
    final_x = np.array([y2,y3])
    finalOutput = AND_logic(final_x)
    return finalOutput

test1 = np.array([0,1])
test2 = np.array([1,1])
test3 = np.array([0,0])
test4 = np.array([1,0])
print(AND_logic(test1))
print(AND_logic(test2))
print(AND_logic(test3))
print(AND_logic(test4),'\n')
```

```
print(OR_logic(test1))
print(OR_logic(test2))
print(OR_logic(test3))
print(OR_logic(test4),'\n')


print(XOR_logic(test1))
print(XOR_logic(test2))
print(XOR_logic(test3))
print(XOR_logic(test4))
```

OUTPUT:

```
0
1
0
0

1
1
0
1

1
0
0
1
```

Reg No:2117010
Program No:04
Date:03-01-2023
Program Name: Simulate Adaline network for AND, OR gates

```
import numpy as np
#Initial Values
INPUTS = np.array([[1,1],[1,-1],[-1,1],[-1,-1]])
LEARNING_RATE = 0.1
#step function
def step_func(sum):
    if sum >= 0:
        return 1
    return -1

#calculating output
def cal_output(weights, instance, bias):
    sum = instance.dot(weights) + bias
    return step_func(sum)

#additing Algorithm
def addline(outputs, weights, bias ):
    total_error = 1
    counter = 0
    while total_error != 0 and counter < 10:
        total_error = 0
        counter += 1
        for i in range(len(outputs)):
            sum = INPUTS[i].dot(weights) + bias
            prediction = step_func(sum)
            total_error += outputs[i] - prediction
            error = outputs[i] – sum

        if outputs[i] != prediction:
            weights[0] = weights[0] + (LEARNING_RATE * error * INPUTS[i][0])
            weights[1] = weights[1] + (LEARNING_RATE * error * INPUTS[i][1])
            bias = bias + (LEARNING_RATE * error)
            print("Weight Updated : " + str(weights[0]))
            print("Weight Updated : " + str(weights[1]))
            print("Bias Updated : " + str(bias))
            print("--------------------------------------------")
    print("Total Error : "+ str(total_error))
    print("--------------------------------------------")
    return weights, bias

if __name__ == "__main__":
    and_outputs = np.array([1,-1,-1,-1])
    or_outputs = np.array([1,1,1,-1])
    weights = np.array([0.0,0.0])
    bias = 0
    returned_weights, returned_bias =  addline(and_outputs, weights, bias)
    print('Prediction for [1,1] : '+ str(cal_output(returned_weights,np.array([1,1]),returned_bias)))
```

```
print('Prediction for [1,-1] : '+ str(cal_output(returned_weights,np.array([1,-1]),returned_bias)))
print('Prediction for [-1,1] : '+ str(cal_output(returned_weights,np.array([-1,1]),returned_bias)))
print('Prediction for [-1,-1] : '+ str(cal_output(returned_weights,np.array([-1,-1]),returned_bias)))
```

OUTPUT:

```
Weight Updated : 0.1
Weight Updated : 0.1
Bias Updated : -0.1
---------------------------------------------
Total Error : 0
---------------------------------------------
Prediction for [1,1] : 1
Prediction for [1,-1] : -1
Prediction for [-1,1] : -1
Prediction for [-1,-1] : -1
```

Reg No:2117010
Program No:05
Date:03-01-2023
Program Name: Simulate Madaline network for AND, OR and XOR gates

---

```
import numpy as np
LEARNING_RATE = 0.1
def step(x):
    if (x >0):
        return 1
    else:
        return -1;

INPUTS1=[-1,-1,1,1]
INPUTS2 =[-1,1,-1,1]
OUTPUTS =[-1,-1,1,-1]
np.random.seed(1)
WEIGHTS11 =0.05
WEIGHTS12 =0.1
WEIGHTS21=0.2
WEIGHTS22=0.2
Bias1=0.3
Bias2=0.15
v1=v2=b3=0.5
errors = []
for i in range(len(INPUTS1)):
    MADALINE_OUTPUT1 = (INPUTS1[i] * WEIGHTS11) + (INPUTS2[i] * WEIGHTS21) + Bias1
    MADALINE_OUTPUT2 = (INPUTS1[i] * WEIGHTS12) + (INPUTS2[i] * WEIGHTS22) + Bias2

    MADALINE_OUTPUT_HIDDEN = (MADALINE_OUTPUT1*v1)+(MADALINE_OUTPUT2*v2)+b3
    MADALINE_OUTPUT_HIDDEN1=step(MADALINE_OUTPUT_HIDDEN)

    if( MADALINE_OUTPUT_HIDDEN1>0):
        WEIGHTS11 = WEIGHTS11 + LEARNING_RATE *(1- MADALINE_OUTPUT_HIDDEN)*
                    INPUTS1[i]
        WEIGHTS21 = WEIGHTS21 + LEARNING_RATE * (1- MADALINE_OUTPUT_HIDDEN)*
                    INPUTS2[i]
        WEIGHTS12 = WEIGHTS12 + LEARNING_RATE * (1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS1[i]
        WEIGHTS22 = WEIGHTS22 + LEARNING_RATE * (1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS2[i]
    else:
        WEIGHTS11 = WEIGHTS11 + LEARNING_RATE * (-1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS1[i]
        WEIGHTS21 = WEIGHTS21 + LEARNING_RATE * (-1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS2[i]
        WEIGHTS12 = WEIGHTS12 + LEARNING_RATE * (-1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS1[i]
        WEIGHTS22 = WEIGHTS22 + LEARNING_RATE * (-1 - MADALINE_OUTPUT_HIDDEN) *
                    INPUTS2[i]
        MADALINE_OUTPUT1 = (INPUTS1[i] * WEIGHTS11) + (INPUTS2[i] * WEIGHTS21) + Bias1
        MADALINE_OUTPUT2 = (INPUTS1[i] * WEIGHTS12) + (INPUTS2[i] * WEIGHTS22) + Bias2
        MADALINE_OUTPUT_HIDDEN = (MADALINE_OUTPUT1 * v1) + (MADALINE_OUTPUT2 *
```

```
                                    v2) + b3
    MADALINE_OUTPUT_HIDDEN1 = step(MADALINE_OUTPUT_HIDDEN)
    print("Actual ", MADALINE_OUTPUT_HIDDEN1, "Desired ", OUTPUTS[i])
```

OUTPUT:

```
Actual  1 Desired  -1
Actual  1 Desired  -1
Actual  1 Desired  1
Actual  1 Desired  -1
```

Reg No:2117010
Program No:06
Date:03-01-2023
Program Name: Simulate Back propagation network

```
import numpy as np
def sigmoid(g):
    return 1/ (1+ np.exp(-2 * g))
def sigmoid_gradient(g):
    return g * (1 - g )
def feedforwardprop(input_layer, output_layer, hidden_weights, output_weights, bias):
    z2 = np.dot(input_layer, hidden_weights)
    a2 = sigmoid(z2)
    a2 = a2.T
    a2 = np.vstack((a2,bias)).T
    z3 = np.dot(a2, output_weights)
    a3 = sigmoid(z3)
    return a2, a3, hidden_weights, output_weights
def backpropogation(input_layer, output_layer, hidden_weights, output_weights, bias, iterations):
    for _ in range(iterations):
        a2, a3, hidden_weights, output_weights = feedforwardprop(input_layer, output_layer, hidden_weights,
output_weights, bias)
        error_a3 = output_layer - a3
        error_a2 = np.dot(error_a3, output_weights[0:2, :].T) * sigmoid(np.dot(input_layer, hidden_weights))
        delta_a3 = error_a3 * sigmoid_gradient(a3)
        delta_a2 = error_a2 * sigmoid_gradient(a2[:, 0:2])
        #updated weights
        output_weights += np.dot(a2.T, delta_a3)
        hidden_weights += np.dot(input_layer.T, delta_a2)
        return a3
#data
input_layer = np.array([[0,0,1], [0,1,1], [1,0,1], [1,1,1]])
output_layer = np.array([[0,1,1,0]]).T
#randomly init..weights
np.random.seed(1)
hidden_weights = np.random.random((3, 2))
output_weights = np.random.random((3, 1))

#number of iteration
iterations = 10000
#bias term
bias = np.ones((1, 4))

print(backpropogation(input_layer, output_layer, hidden_weights, output_weights, bias, iterations))
```

OUTPUT:

```
[[0.799683  ]
 [0.81490972]
 [0.83921297]
 [0.84540002]]
```

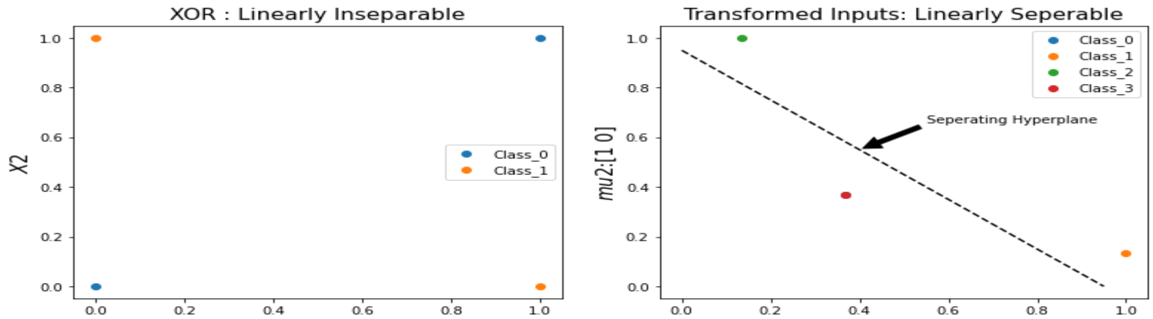
Register No:2117010
Program No:07
Date:03-01-2023
Program Name: Implement Radial Basis Function for XOR gates

```
import numpy as np
import matplotlib.pyplot as plt
def gaussian_rbf(x, landmark, gamma=1):
    return np.exp(-gamma * np.linalg.norm(x - landmark)**2)
#solving prblm in matrices form
def end_to_end(X1, X2, ys, mu1, mu2):
    from_1 = [gaussian_rbf(i,mu1) for i in zip(X1,X2)]
    from_2 = [gaussian_rbf(i,mu2) for i in zip(X1,X2)]
    plt.figure(figsize=(13,5))
    plt.subplot(1,2,1)
    plt.scatter((x1[0],x1[3]),(x2[0],x2[3]), label = "Class_0")
    plt.scatter((x1[1],x1[2]),(x2[1],x2[2]), label = "Class_1")
    plt.xlabel("$X1$", fontsize=15)
    plt.ylabel("$X2$", fontsize=15)
    plt.title("XOR : Linearly Inseparable", fontsize=15)
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.scatter(from_1[0], from_2[0], label="Class_0")
    plt.scatter(from_1[1], from_2[1], label="Class_1")
    plt.scatter(from_1[2], from_2[2], label="Class_2")
    plt.scatter(from_1[3], from_2[3], label="Class_3")
    plt.plot([0, 0.95],[0.95,0], "k--")
    plt.annotate("Seperating Hyperplane", xy=(0.4,0.55),
xytext=(0.55,0.66),arrowprops=dict(facecolor='black', shrink=0.05))
    plt.xlabel(f"$mu1$:{(mu1)}", fontsize=15)
    plt.ylabel(f"$mu2$:{(mu2)}", fontsize=15)
    plt.title("Transformed Inputs: Linearly Seperable", fontsize=15)
    plt.legend()
    A = []
    for i, j in zip(from_1, from_2):
        temp=[]
        temp.append(i)
        temp.append(j)
        temp.append(1)
        A.append(temp)
    A = np.array(A)
    W = np.linalg.inv(A.T.dot(A)).dot(A.T).dot(ys)
    print(np.round(A.dot(W)))
    print(ys)
    print(f"Weights : {W}")
    return W
def predict_matrix(point, weights):
    gaussian_rbf_0 = gaussian_rbf(np.array(point), mu1)
    gaussian_rbf_1 = gaussian_rbf(np.array(point), mu2)
    A = np.array([gaussian_rbf_0, gaussian_rbf_1, 1])
    return np.round(A.dot(weights))
#points
x1 = np.array([0,0,1,1])
```

```
x2 = np.array([0,1,0,1])
ys = np.array([0,1,1,0])
#centers
mu1 = np.array([0,1])
mu2 = np.array([1,0])
w = end_to_end(x1,x2,ys,mu1,mu2)
#testing
print(f"Input:{np.array([0,0])}, Predicted: {predict_matrix(np.array([0,0]), w)}")
print(f"Input:{np.array([0,1])}, Predicted: {predict_matrix(np.array([0,1]), w)}")
print(f"Input:{np.array([1,0])}, Predicted: {predict_matrix(np.array([1,0]), w)}")
print(f"Input:{np.array([1,1])}, Predicted: {predict_matrix(np.array([1,1]), w)}")
```

OUTPUT:

```
[0. 1. 1. 0.]
[0 1 1 0]
Weights : [ 2.5026503   2.5026503  -1.84134719]
Input:[0 0], Predicted: 0.0
Input:[0 1], Predicted: 1.0
Input:[1 0], Predicted: 1.0
Input:[1 1], Predicted: 0.0
```



Register No:2117010
Program No:08
Date:03-01-2023
Program Name: Simulate Kohonen SOM algorithm

```
import math
class SOM:
    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)
            D0 = 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
#driver code
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 = SOM()
    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:

```
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.6000000000000001, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.30000000000000004, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.65, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.325, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.6625000000000001, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.33125000000000004, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.665625, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.3328125, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.6664062500000001, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.33320312500000004, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.6666015625, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
Test Sample s belongs to Cluster :  0
Trained Weights :  [[0.33330078125, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
```

REGISTER NUMBER:2117010
PROGRAM NO:09
Date:16-01-2023
Program Name: Simulate Learning Vector Quantization (LVQ)

```
import math as M
class LVQ :
   def winner1( self, weights, sample ) :
      D_0 = 0
      D_1 = 0
      for K in range( len( sample ) ) :
         D_0 = D_0 + M.pow( ( sample[K] - weights[0][K] ), 2 )
         D_1 = D_1 + M.pow( ( sample[K] - weights[1][K] ), 2 )
         if D_0 > D_1:
            return 0
         else :
            return 1
   def update1( self, weights, sample, J, alpha1 ) :
      for k in range(len(weights)) :
         weights[J][k] = weights[J][k] + alpha1 * ( sample[k] - weights[J][k] )
def main() :
   P = [[ 0, 0, 1, 1 ], [ 1, 0, 1, 0 ],  [ 0, 0, 0, 1 ], [ 0, 1, 1, 0 ],  [ 1, 1, 1, 0 ], [ 1, 0, 1, 1 ],]
   Q = [ 0, 1, 0, 1, 0, 1 ]
   g, h = len( P ), len( P[0] )
   weights = []
   weights.append( P.pop( 0 ) )
   weights.append( P.pop( 1 ) )
   g = g - 2
   ob1 = LVQ()
   epochs1 = 3
   alpha1 = 0.1
   for k in range( epochs1 ):
      for o in range( g ) :
         T = P[o]
         J = ob1.winner1( weights, T )
         ob1.update1( weights, T, J, alpha1 )
   T = [ 0, 1, 1, 0 ]
   J = ob1.winner1( weights, T )
   print( "Sample T belongs to class : ", J )
   print( "Trained weights : ", weights )

if __name__ == "__main__":
   main()
```

OUTPUT:

```
Sample T belongs to class :  0
Trained weights :  [[0.3660931, 0.2816541, 1, 1], [0.33661, 0.1729, 0, 1]]
```

Register No:2117010
Program No:10
Date:03-01-2023
Program Name: Simulate Hamming network

```
def main():
    e1 = [-1, 1, -1, 1]
    e2 = [1, -1, -1, -1]
    p1 = [1, 1, -1, -1]
    p2 = [-1, 1, 1, -1]
    p3 = [-1, -1, -1, 1]
    p4 = [-1, -1, 1, 1]
    print("E1 :", e1, "\tE2 :", e2)
    print("P1 :", p1, "\tP2 :", p2)
    print("P3 :", p3, "\tP4 :", p4)
    w = [[-0.5, 0.5], [0.5, -0.5], [-0.5, -0.5], [0.5, -0.5]]
    display(" W")
    display(w)
    b1 = b2 = 4 / 2
    print("b1 = b2 = N/2 :", b1)
    yin1 = yin2 = 0
    print("\n Net Input for vector 1")
    for i in range(len(w)):
        yin1 += p1[i] * w[i][0]
    yin1 += b1
    print(" Yin1 :", yin1)

    for i in range(len(w)):
        yin2 += p1[i] * w[i][1]
    yin2 += b2
    print(" Yin2 :", yin2)
    print(" y2(0) > (0) > y1(0),")
    print(" [1,1,-1,-1]")
    yin1 = yin2 = 0
    print("\n Net Input for vector 2")

    for i in range(len(w)):
        yin1 += p2[i] * w[i][0]
    yin1 += b1
    print(" Yin1 :", yin1)

    for i in range(len(w)):
        yin2 += p2[i] * w[i][1]
    yin2 += b2
    print(" Yin2 :", yin2)
    print(" y1(0) > y2(0),")
    print(" [-1,1,1,-1]")
    yin1 = yin2 = 0
    print("\n Net Input for vector 3")

    for i in range(len(w)):
        yin1 += p3[i] * w[i][0]
    yin1 += b1
```

```
    print(" Yin1 :", yin1)

    for i in range(len(w)):
        yin2 += p3[i] * w[i][1]
    yin2 += b2
    print(" Yin2 :", yin2)
    print(" y1(0) > y2(0),")
    print(" [-1,-1,-1,1]")
    yin1 = yin2 = 0
    print("\n Net Input for vector 4")

    for i in range(len(w)):
        yin1 += p4[i] * w[i][0]
    yin1 += b1
    print(" Yin1 :", yin1)

    for i in range(len(w)):
        yin2 += p4[i] * w[i][1]
    yin2 += b2
    print(" Yin2 :", yin2)
    print(" y1(0) > y2(0),")
    print(" [-1,-1,1,1]")
def display(m):
    for i in m:
        print(i)
main()
```

OUTPUT:

```
E1 : [-1, 1, -1, 1]     E2 : [1, -1, -1, -1]
P1 : [1, 1, -1, -1]     P2 : [-1, 1, 1, -1]
P3 : [-1, -1, -1, 1]    P4 : [-1, -1, 1, 1]

W
[-0.5, 0.5]
[0.5, -0.5]
[-0.5, -0.5]
[0.5, -0.5]
b1 = b2 = N/2 : 2.0

 Net Input for vector 1
 Yin1 : 2.0
 Yin2 : 3.0
 y2(0) > (0) > y1(0),
 [1,1,-1,-1]

 Net Input for vector 2
 Yin1 : 2.0
 Yin2 : 1.0
 y1(0) > y2(0),
 [-1,1,1,-1]

 Net Input for vector 3
 Yin1 : 3.0
 Yin2 : 2.0
 y1(0) > y2(0),
 [-1,-1,-1,1]


 Net Input for vector 4
 Yin1 : 2.0
 Yin2 : 1.0
 y1(0) > y2(0),
 [-1,-1,1,1]
```

Register No:2117010
Program No:11
Date:03-01-2023
Program Name: Implement Auto Associative Neural Network

```
import numpy as np
import random

print("Auto Associative Networks")
n = int(input('Enter n:'))
X = [ random.choice([-1,1]) for i in range(n)]
Y = [ random.choice([-1,1]) for i in range(n)]
print("Input Vector is",X)
print("Output Vector is",Y)
weights = [ [ 0 for _ in range(n)] for _ in range(n)]
## Training Phase
for i in range(n):
    for j in range(n):
        weights[i][j]+=X[i]*Y[j]
print("Weights after Training:")
print(weights)
## Testing Phase
test = [ random.choice([-1,1]) for i in range(n)]
print("Test Input",test)
def f(yinj):
    if yinj > 0:
        return 1
    else:
        return -1
outs= []
for j in range(n):
    yinj = 0
    for i in range(n):
        yinj+=test[i]*weights[i][j]
    yin = f(yinj)
    outs.append(yin)
print("Testing Output",outs)
```

OUTPUT:

```
Auto Associative Networks
Enter n:3
Input Vector is [-1, 1, -1]
Output Vector is [-1, 1, 1]
Weights after Training:
[[1, -1, -1], [-1, 1, 1], [1, -1, -1]]
Test Input [1, 1, -1]
Testing Output [-1, 1, 1]
```

Register No:2117010
Program No:12
Date:03-01-2023
Program Name: : Implement Heteroassociative Networks

```python
import random
print("Hetero Associative Networks")
n,m = map(int, input("Enter n and m:").split())
X = [ random.choice([-1,1]) for i in range(n)]
Y = [ random.choice([-1,1]) for i in range(m)]
print("Input Vector is",X)
print("Output Vector is",Y)
weights = [ [ 0 for _ in range(m)] for _ in range(n)]
## Training Phase
for i in range(n):
    for j in range(m):
        weights[i][j]+=X[i]*Y[j]
print("Weights after Training:")
print(weights)
## Testing Phase
test = [ random.choice([-1,1]) for i in range(n)]
print("Test Input",test)
def f(yinj):
    if yinj > 0:
        return 1
    else:
        return -1
outs= []
for j in range(m):
    yinj = 0
    for i in range(n):
        yinj+=test[i]*weights[i][j]
    yin = f(yinj)
    outs.append(yin)
print("Testing Output",outs)
```

OUTPUT :

```
Hetero Associative Networks
Enter n and m:2 3
Input Vector is [1, 1]
Output Vector is [-1, 1, 1]
Weights after Training:
[[-1, 1, 1], [-1, 1, 1]]
Test Input [1, 1]
Testing Output [-1, 1, 1]
```

Register No:2117010
Program No:13
Date:03-01-2023
Program Name: Bidirectional Associative Memory (BAM) Implementation

---

```
# Bidirectional Associative Memory (BAM) Implementation
import numpy as np

# Take two sets of patterns:
# Set A: Input Pattern
x1 = np.array([1, 1, 1, 1, 1, 1]).reshape(6, 1)
x2 = np.array([-1, -1, -1, -1, -1, -1]).reshape(6, 1)
x3 = np.array([1, 1, -1, -1, 1, 1]).reshape(6, 1)
x4 = np.array([-1, -1, 1, 1, -1, -1]).reshape(6, 1)

# Set B: Target Pattern
y1 = np.array([1, 1, 1]).reshape(3, 1)
y2 = np.array([-1, -1, -1]).reshape(3, 1)
y3 = np.array([1, -1, 1]).reshape(3, 1)
y4 = np.array([-1, 1, -1]).reshape(3, 1)


print("Set A: Input Pattern, Set B: Target Pattern")
print("\nThe input for pattern 1 is")
print(x1)
print("\nThe target for pattern 1 is")
print(y1)
print("\nThe input for pattern 2 is")
print(x2)
print("\nThe target for pattern 2 is")
print(y2)
print("\nThe input for pattern 3 is")
print(x3)
print("\nThe target for pattern 3 is")
print(y3)
print("\nThe input for pattern 4 is")
print(x4)
print("\nThe target for pattern 4 is")
print(y4)

print("\n------------------------------")

# Calculate weight Matrix: W
inputSet = np.concatenate((x1, x2, x3, x4), axis = 1)
targetSet = np.concatenate((y1.T, y2.T, y3.T, y4.T), axis = 0)
print("\nWeight matrix:")
weight = np.dot(inputSet, targetSet)
print(weight)
```

```
print("\n-----------------------------")

# Testing Phase
# Test for Input Patterns: Set A
print("\nTesting for input patterns: Set A")
def testInputs(x, weight):
    # Multiply the input pattern with the weight matrix
    # (weight.T X x)
    y = np.dot(weight.T, x)
    y[y < 0] = -1
    y[y >= 0] = 1
    return np.array(y)

print("\nOutput of input pattern 1")
print(testInputs(x1, weight))
print("\nOutput of input pattern 2")
print(testInputs(x2, weight))
print("\nOutput of input pattern 3")
print(testInputs(x3, weight))
print("\nOutput of input pattern 4")
print(testInputs(x4, weight))

# Test for Target Patterns: Set B
print("\nTesting for target patterns: Set B")
def testTargets(y, weight):
    # Multiply the target pattern with the weight matrix
    # (weight X y)
    x = np.dot(weight, y)
    x[x <= 0] = -1
    x[x > 0] = 1
    return np.array(x)

print("\nOutput of target pattern 1")
print(testTargets(y1, weight))
print("\nOutput of target pattern 2")
print(testTargets(y2, weight))
print("\nOutput of target pattern 3")
print(testTargets(y3, weight))
print("\nOutput of target pattern 4")
print(testTargets(y4, weight))
```

OUTPUT :

```
Set A: Input Pattern, Set B: Target Pattern

The input for pattern 1 is
[[1]
 [1]
 [1]
 [1]
 [1]
 [1]]

The target for pattern 1 is
[[1]
 [1]
 [1]]

The input for pattern 2 is
[[-1]
 [-1]
 [-1]
 [-1]
 [-1]
 [-1]]

The target for pattern 2 is
[[-1]
 [-1]
 [-1]]

The input for pattern 3 is
[[ 1]
 [ 1]
 [-1]
 [-1]
 [ 1]
 [ 1]]

The target for pattern 3 is
[[ 1]
 [-1]
 [ 1]]

The input for pattern 4 is
[[-1]
 [-1]
 [ 1]
 [ 1]
 [-1]
 [-1]]

The target for pattern 4 is
[[-1]
 [ 1]
 [-1]]
```

```
Testing for input patterns: Set A

Output of input pattern 1
[[1]
 [1]
 [1]]

Output of input pattern 2
[[-1]
 [-1]
 [-1]]

Output of input pattern 3
[[ 1]
 [-1]
 [ 1]]

Output of input pattern 4
[[-1]
 [ 1]
 [-1]]

Testing for target patterns: Set B

Output of target pattern 1
[[1]
 [1]
 [1]
 [1]
 [1]
 [1]]

Output of target pattern 2
[[-1]
 [-1]
 [-1]
 [-1]
 [-1]
 [-1]]

Output of target pattern 3
[[ 1]
 [ 1]
 [-1]
 [-1]
 [ 1]
 [ 1]]
```

```
Output of target pattern 4
[[-1]
 [-1]
 [ 1]
 [ 1]
 [-1]
 [-1]]
```