Artificial Neural Networks Practical File

Shubh Gupta (2019UIT3080), Naman Singh (2019UIT3086), Yajas Sardana (2019UIT3099), Arhant Jain (2019UIT3104), Dhruv Tiwari (2019UIT3115), Himanshu Gupta (2019UIT3116) - [IT 2]

Ques1 - Realize MCP model for logic OR gate.

Code:-

```
class MpNeuron:
  inpOuts = []

  def addinpOuput(self, inp, out):
    self.inpOuts.append([inp, out])

  def getNetinp(self, inp, weights):
    netinp = 0
    for i in range(len(inp)):
      netinp += inp[i] * weights[i]
    return netinp

  def getValidThreshold(self, weights):
    threshold = 10000000

    for inp, out in self.inpOuts:
      if out == 1:
        threshold = min(threshold, self.getNetinp(inp, weights))

    for inp, out in self.inpOuts:
      if out == 0 and threshold <= self.getNetinp(inp, weights):
        return None

    return threshold


orNeuron = MpNeuron()
orNeuron.addinpOuput([0, 0], 0)
orNeuron.addinpOuput([0, 1], 1)
orNeuron.addinpOuput([1, 0], 1)
orNeuron.addinpOuput([1, 1], 1)

print('Or Gate\'s Truth Table: ')
print('x1 x2 y')
print('-------')
```

```
print('0  0  0')
print('0  1  1')
print('1  0  1')
print('1  1  1')
print()

while True:
  weights = list(map(int, input('Enter weights: ').split(' ')))
  threshold = orNeuron.getValidThreshold(weights)

  if threshold != None:
    print('Weights are correct, Threshold Found:', threshold)
    print('Hence, and gate can be realised using mp neuron')
    break
  else:
    print('Invalid weights')
    print()
```

Output:-

```
Or Gate's Truth Table:
x1 x2 y
-------
0  0  0
0  1  1
1  0  1
1  1  1

Enter weights: 1 1
Weights are correct, Threshold Found: 1
Hence, and gate can be realised using mp neuron
```

Ques2 - Realize MCP model for logic XOR gate.

Code:-

```
class MpNeuron:
 def __init__(self):
  self.inpOuts = []
  self.weights = []
  self.threshold = []

 def addinpOuput(self, inp, out):
  self.inpOuts.append([inp, out])

 def getNetinp(self, inp, weights):
  netinp = 0
  for i in range(len(inp)):
   netinp += inp[i] * weights[i]
  return netinp

 def checkAndSetWeights(self, weights):
  threshold = 10000000
  # print(weights)

  for inp, out in self.inpOuts:
   if out == 1:
    threshold = min(threshold, self.getNetinp(inp, weights))

  for inp, out in self.inpOuts:
   if out == 0 and threshold <= self.getNetinp(inp, weights):
    return False

  self.weights = weights
  self.threshold = threshold
  return True

 def validWeightsCalc(self, low, high):
  for w1 in range(low, high + 1):
   for w2 in range(low, high + 1):
    if self.checkAndSetWeights([w1, w2]):
     return True
  return False

 def getCalculatedOutput(self, inp):
  if self.getNetinp(inp, self.weights) >= self.threshold:
```

```
      return 1
    return 0


andNotNeuron = MpNeuron()
andNotNeuron.addinpOuput([0, 0], 0)
andNotNeuron.addinpOuput([0, 1], 0)
andNotNeuron.addinpOuput([1, 0], 1)
andNotNeuron.addinpOuput([1, 1], 0)
andNotNeuron.validWeightsCalc(-1, 1)
print('For andNot, ([w1, w2], theta): ',
      andNotNeuron.weights, andNotNeuron.threshold)


andNotReverseNeuron = MpNeuron()
andNotReverseNeuron.addinpOuput([0, 0], 0)
andNotReverseNeuron.addinpOuput([0, 1], 1)
andNotReverseNeuron.addinpOuput([1, 0], 0)
andNotReverseNeuron.addinpOuput([1, 1], 0)
andNotReverseNeuron.validWeightsCalc(-1, 1)
print('For andNotReverse, ([w1, w2], theta): ',
      andNotReverseNeuron.weights, andNotReverseNeuron.threshold)

orNeuron = MpNeuron()
orNeuron.addinpOuput([0, 0], 0)
orNeuron.addinpOuput([0, 1], 1)
orNeuron.addinpOuput([1, 0], 1)
orNeuron.addinpOuput([1, 1], 1)
orNeuron.validWeightsCalc(-2, 2)
print('For or, ([w1, w2], theta): ',
      orNeuron.weights, orNeuron.threshold)

print('')
print('Xor Neuron:')
print('i1', 'i2', 'out')
for i1 in range(0, 2):
  for i2 in range(0, 2):
    print(i1, i2, orNeuron.getCalculatedOutput(
        [andNotNeuron.getCalculatedOutput([i1, i2]),
andNotReverseNeuron.getCalculatedOutput([i1, i2])]))
```

Output:-

```
For andNot, ([w1, w2], theta):  [1, -1] 1
For andNotReverse, ([w1, w2], theta):  [-1, 1] 1
For or, ([w1, w2], theta):  [1, 1] 1

Xor Neuron:
i1 i2 out
0 0 0
0 1 1
1 0 1
1 1 0
```

Ques3 - Implement Hebb learning rule for AND logic gate.

Code:-

```
def hebb_and():
    X = [[1, 1], [1, -1], [-1, 1], [-1, -1]]
    Y = [1, -1, -1, -1]
    weights = [0, 0]
    bias = 0
    check = False
    while not check:
        for i in [0, 1, 2, 3]:
            weights[0] = weights[0]+X[i][0]*Y[i]
            weights[1] = weights[1]+X[i][1]*Y[i]
            bias = Y[i]
        check = True
        for i in [0, 1, 2, 3]:
            if func(weights[0]*X[i][0] + weights[1]*X[i][1] + bias) != Y[i]:
                check = False
    print("W1 is :-", weights[0])
    print("W2 is :-", weights[1])
    print("Bias is :-", bias)
```

```
def func(a):
    if(a > 0):
        return 1
    else:
        return -1


hebb_and()
```

Output:-

```
W1 is :- 2
W2 is :- 2
Bias is :- -1
```

Ques4 - Implement Perceptron learning rule for NOT AND Logic gate.

Code:-

```
def activationFn(x, theta):
  if x < -theta:
    return -1
  if -theta <= x <= theta:
    return 0
  return 1


class Perceptron:
  def __init__(self, N, theta=0, alpha=1):
    self.N = N
    self.weights = [0] * N
    self.bias = 0
    self.theta = theta
```

```
  self.alpha = alpha
  self.expectedInOuts = []

def addexpectedInOut(self, inp, out):
  self.expectedInOuts.append([inp, out])

def getNetinp(self, inp):
  netinp = self.bias
  for i in range(len(inp)):
    netinp += inp[i] * self.weights[i]
  return netinp

def getNetOut(self, inp):
  return activationFn(self.getNetinp(inp), self.theta)

# eg - x1 x2 1  t  yin  y      dw1 dw2 db   w1 w2 b
def printLine(self, x, t, yin, y, dw, db, w, b):
  def formattedStr(str): return '{:>4}'.format(str)

  s = ''
  for val in x:
    s += formattedStr(val)
  s += formattedStr(1)
  s += '  |  '
  s += formattedStr(t)
  s += '  |  '
  s += formattedStr(yin)
  s += '  |  '
  s += formattedStr(y)
  s += '  |  '
  for val in dw:
    s += formattedStr(val)
  s += formattedStr(db)
  s += '  |  '
  for val in w:
    s += formattedStr(val)
  s += formattedStr(b)

  print(s)
```

```
def train(self):
  changed = True
  epoch = 1

  self.printLine(
      ['x{}'.format(i) for i in range(1, self.N + 1)],
      't',
      'yin',
      'y',
      ['dw{}'.format(i) for i in range(1, self.N + 1)],
      'db',
      ['w{}'.format(i) for i in range(1, self.N + 1)],
      'b'
  )

  while changed:
    changed = False
    print('EPOCH -', epoch)

    for x, t in self.expectedInOuts:
      yin = self.getNetinp(x)
      y = self.getNetOut(x)
      dw = [0] * self.N
      db = 0
      if y != t:
        for i in range(self.N):
          dw[i] = self.alpha * t * x[i]
          self.weights[i] += dw[i]
        db = self.alpha * t
        self.bias += db
        changed = True

      self.printLine(x, t, yin, y, dw, db, self.weights, self.bias)

    epoch += 1

def test(self, inp):
  return self.getNetOut(inp)
```

```
andNot = Perceptron(2, theta=0, alpha=1)
andNot.addexpectedInOut([1, 1], 1)
andNot.addexpectedInOut([1, -1], -1)
andNot.addexpectedInOut([-1, 1], 1)
andNot.addexpectedInOut([-1, -1], 1)

andNot.train()
print(andNot.weights, andNot.bias)
```

Output:-

```
  x1  x2   1  |     t  |   yin  |     y  |   dw1 dw2  db  |    w1  w2   b
EPOCH - 1
   1   1   1  |     1  |     0  |     0  |     1   1   1  |     1   1   1
   1  -1   1  |    -1  |     1  |     1  |    -1   1  -1  |     0   2   0
  -1   1   1  |     1  |     2  |     1  |     0   0   0  |     0   2   0
  -1  -1   1  |     1  |    -2  |    -1  |    -1  -1   1  |    -1   1   1
EPOCH - 2
   1   1   1  |     1  |     1  |     1  |     0   0   0  |    -1   1   1
   1  -1   1  |    -1  |    -1  |    -1  |     0   0   0  |    -1   1   1
  -1   1   1  |     1  |     3  |     1  |     0   0   0  |    -1   1   1
  -1  -1   1  |     1  |     1  |     1  |     0   0   0  |    -1   1   1
[-1, 1] 1
```

Ques5 - Use Adaline networks to implement AND/OR Logic gate.

OR -

Code:-

```
import numpy as np

features = np.array(
    [
        [-1, -1],
        [-1, 1],
        [1, -1],
```

```
        [1, 1]
    ])

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

print(features, labels)

weight = [0, 0]
bias = 1
learning_rate = 0.2
epoch = 10

for i in range(epoch):

    print("epoch :", i+1)

    sum_squared_error = 0.0

    for j in range(features.shape[0]):

        actual = labels[j]

        x1 = features[j][0]
        x2 = features[j][1]

        unit = (x1 * weight[0]) + (x2 * weight[1]) + bias

        error = actual - unit

        print("error =", error)

        sum_squared_error += error * error

        weight[0] += learning_rate * error * x1
        weight[1] += learning_rate * error * x2

        bias += learning_rate * error

    print("sum of squared error = ", sum_squared_error/4, "\n\n")
```

Output:-

```
[[-1 -1]
 [-1  1]
 [ 1 -1]
 [ 1  1]] [-1  1  1  1]
epoch : 1
error = -2
error = 0.4
error = 0.4800000000000001
error = -0.5760000000000001
sum of squared error =  1.1805440000000003


epoch : 2
error = -1.0912000000000002
error = 0.58944
error = 0.6433279999999999
error = -0.6951936000000001
sum of squared error =  0.6088305026662401


epoch : 3
error = -0.8220723199999999
error = 0.6678947840000002
error = 0.6943633408000001
error = -0.7149435289600001
sum of squared error =  0.5287927601133261


epoch : 4
error = -0.744269258752
error = 0.6978731393024
error = 0.7091625217228802
error = -0.7162383955539456
sum of squared error =  0.5142181423860819
```

AND -

Code:-

```
import numpy as np

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

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

print(features, labels)

weight = [0, 0]
bias = 1
learning_rate = 0.2
epoch = 10

for i in range(epoch):

    print("epoch :", i+1)

    sum_squared_error = 0.0

    for j in range(features.shape[0]):

        actual = labels[j]

        x1 = features[j][0]
        x2 = features[j][1]

        unit = (x1 * weight[0]) + (x2 * weight[1]) + bias

        error = actual - unit

        print("error =", error)

        sum_squared_error += error * error
```

```
        weight[0] += learning_rate * error * x1
        weight[1] += learning_rate * error * x2

        bias += learning_rate * error

    print("sum of squared error = ", sum_squared_error/4, "\n\n")
```

Output:-

```
[[-1 -1]
 [-1  1]
 [ 1 -1]
 [ 1  1]] [-1 -1 -1  1]
epoch : 1
error = -2
error = -1.6
error = -1.92
error = 0.30400000000000005
sum of squared error =  2.5847040000000003


epoch : 2
error = -0.03519999999999999
error = -1.0777599999999996
error = -1.037312
error = 0.5375744000000001
sum of squared error =  0.6319520196198398


epoch : 3
error = 0.51644928
error = -0.849371136
error = -0.7956037631999999
error = 0.64731459584
sum of squared error =  0.5100381798719882


epoch : 4
error = 0.6650376110080001
error = -0.7613396484096001
error = -0.73297987633152
error = 0.690797265485824
sum of squared error =  0.509093361351336
```

Ques6 - Implement backpropagation algorithm to classify a dataset. (Select a binary classification dataset from UCI repository).

Code:-

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
# here planar_utils.py can be found on its github repo
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset
# Loading the Sample data
X, Y = load_planar_dataset()

# Visualize the data:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral)

# X --> input dataset of shape (input size, number of examples)
# Y --> labels of shape (output size, number of examples)

W1 = np.random.randn(4, X.shape[0]) * 0.01
b1 = np.zeros(shape=(4, 1))

W2 = np.random.randn(Y.shape[0], 4) * 0.01
b2 = np.zeros(shape=(Y.shape[0], 1))


def forward_prop(X, W1, W2, b1, b2):

  Z1 = np.dot(W1, X) + b1
  A1 = np.tanh(Z1)
  Z2 = np.dot(W2, A1) + b2
  A2 = sigmoid(Z2)

  # here the cache is the data of previous iteration
  # This will be used for backpropagation
  cache = {"Z1": Z1,
           "A1": A1,
           "Z2": Z2,
           "A2": A2}
```

```
  return A2, cache

# Here Y is actual output


def compute_cost(A2, Y):
  m = Y.shape[1]
  # implementing the above formula
  cost_sum = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1 - A2))
  cost = - np.sum(cost_sum) / m

  # Squeezing to avoid unnecessary dimensions
  cost = np.squeeze(cost)
  return cost


learning_rate = 0.1


def back_prop(W1, b1, W2, b2, cache):
  m = Y.shape[1]

  # Retrieve also A1 and A2 from dictionary "cache"
  A1 = cache['A1']
  A2 = cache['A2']

  # Backward propagation: calculate dW1, db1, dW2, db2.
  dZ2 = A2 - Y
  dW2 = (1 / m) * np.dot(dZ2, A1.T)
  db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

  dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
  dW1 = (1 / m) * np.dot(dZ1, X.T)
  db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

  # Updating the parameters according to algorithm
  W1 = W1 - learning_rate * dW1
  b1 = b1 - learning_rate * db1
  W2 = W2 - learning_rate * dW2
  b2 = b2 - learning_rate * db2
```

```
  return W1, W2, b1, b2


num_iterations = 10000
# Please note that the weights and bias are global
# Here num_iteration is epochs
for i in range(0, num_iterations):

  # Forward propagation. Inputs: "X, parameters". return: "A2, cache".
  A2, cache = forward_prop(X, W1, W2, b1, b2)

  # Cost function. Inputs: "A2, Y". Outputs: "cost".
  cost = compute_cost(A2, Y)

  # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
  W1, W2, b1, b2 = back_prop(W1, b1, W2, b2, cache)

  # Print the cost every 1000 iterations
  if cost and i % 1000 == 0:
    print("Cost after iteration % i: % f" % (i, cost))
```

Output:-

```
Cost after iteration  0:  0.693113
Cost after iteration  1000:  0.420077
Cost after iteration  2000:  0.332179
Cost after iteration  3000:  0.315812
Cost after iteration  4000:  0.307509
Cost after iteration  5000:  0.302026
Cost after iteration  6000:  0.297915
Cost after iteration  7000:  0.294589
Cost after iteration  8000:  0.291767
Cost after iteration  9000:  0.289302
```

Ques7 - Implement Autoassociative and Heteroassociative memory for pattern association.

Autoassociative -

Code:-

```
print("Auto Associative Networks::")

t = int(input("Enter number of input samples: "))
n = int(input("Enter the number of nodes: "))

X = []
for i in range(t):          # A for loop for row entries
    a = list(map(int, input().split()))
    X.append(a)

Y = X

print("Input Vector is", X)
print("Output Vector is", Y)

weights = [[0 for _ in range(n)] for _ in range(n)]

# Training Phase
for k in range(t):
    for i in range(n):
        for j in range(n):
            weights[i][j] += X[k][i]*Y[k][j]

print("Weights after Training:")
print(weights)

print("Enter the testing vector: ")
# Testing Phase
test = list(map(int, input().split()))

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 number of input samples: 2
Enter the number of nodes: 4
1 1 -1 1
1 -1 -1 -1
Input Vector is [[1, 1, -1, 1], [1, -1, -1, -1]]
Output Vector is [[1, 1, -1, 1], [1, -1, -1, -1]]
Weights after Training:
[[2, 0, -2, 0], [0, 2, 0, 2], [-2, 0, 2, 0], [0, 2, 0, 2]]
Enter the testing vector:
1 1 -1 -1
Test Input [1, 1, -1, -1]
Testing Output [1, -1, -1, -1]
```

Heteroassociative -
Code:-

```
print("Heteroassociative Networks::")
```

```
t = int(input("Enter number of input samples: "))
n = int(input("Enter the number of features: "))
n2 = int(input("Enter length of output: "))
X = []
Y = []
for i in range(t):          # A for loop for row entries
    a = list(map(int, input().split()))
    b = list(map(int, input().split()))
    X.append(a)
    Y.append(b)


print("Input Vector is", X)
print("Output Vector is", Y)

weights = [[0 for _ in range(n2)] for _ in range(n)]

# Training Phase
for k in range(t):
    for i in range(n):
        for j in range(n2):
            weights[i][j] += X[k][i]*Y[k][j]

print("Weights after Training:")
print(weights)


def f(yinj):
    if yinj > 0:
        return 1
    if yinj == 0:
        return 0
    else:
        return -1


# Testing Phase
num = int(input("Enter number of test cases: "))
for p in range(num):
```

```
print("Enter the testing vector: ")

test = list(map(int, input().split()))
print("Test Input", test)
outs = []
for j in range(n2):
    yinj = 0
    for i in range(n):
        yinj += test[i]*weights[i][j]
    yin = f(yinj)
    outs.append(yin)

print("Testing Output", outs)
```

Output:-

```
Heteroassociative Networks::

Enter number of input samples: 2

Enter the number of features: 4

Enter length of output: 2

1 1 1 1

1 -1

1 -1 1 -1

-1 1
Input Vector is [[1, 1, 1, 1], [1, -1, 1, -1]]
Output Vector is [[1, -1], [-1, 1]]
Weights after Training:
[[0, 0], [2, -2], [0, 0], [2, -2]]
```

```
Enter the testing vector:

1 1 1 1
Test Input [1, 1, 1, 1]
Testing Output [1, -1]
Enter the testing vector:

1 -1 1 -1
Test Input [1, -1, 1, -1]
Testing Output [-1, 1]
Enter the testing vector:

1 1 1 0
Test Input [1, 1, 1, 0]
Testing Output [1, -1]
Enter the testing vector:
```

Ques8 - Implement bidirectional associative memory for pattern association.

Code:-

```
# Import Python Libraries
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
```

```python
    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:-

```
Weight matrix:
[[4 0 4]
 [4 0 4]
 [0 4 0]
 [0 4 0]
 [4 0 4]
 [4 0 4]]

------------------------------

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]
 [-1]]
```

Ques9 - Implement Hopfield network for any given problem.

Code:-

```
import matplotlib.pyplot as plt
import numpy as np

nb_patterns = 4   # Number of patterns to learn
pattern_width = 5
pattern_height = 5
max_iterations = 10

# Define Patterns
patterns = np.array([
   [1, -1, -1, -1, 1, 1, -1, 1, 1, -1, 1, -1, 1, 1, -1,
      1, -1, 1, 1, -1, 1, -1, -1, -1, 1.],   # Letter D
   [-1, -1, -1, -1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, - \
    1, 1, 1, -1, 1, -1, -1, -1, 1, 1.],    # Letter J
   [1, -1, -1, -1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, - \
    1, 1, 1, 1, 1, 1, -1, -1, -1, -1.],     # Letter C
   [-1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1.], ],  # Letter
M
   dtype=np.float)
fig, ax = plt.subplots(1, nb_patterns, figsize=(15, 10))

for i in range(nb_patterns):
   ax[i].matshow(patterns[i].reshape(
      (pattern_height, pattern_width)), cmap='gray')
   ax[i].set_xticks([])
   ax[i].set_yticks([])
W = np.zeros((pattern_width * pattern_height, pattern_width * pattern_height))

for i in range(pattern_width * pattern_height):
   for j in range(pattern_width * pattern_height):
      if i == j or W[i, j] != 0.0:
         continue

      w = 0.0

      for n in range(nb_patterns):
```

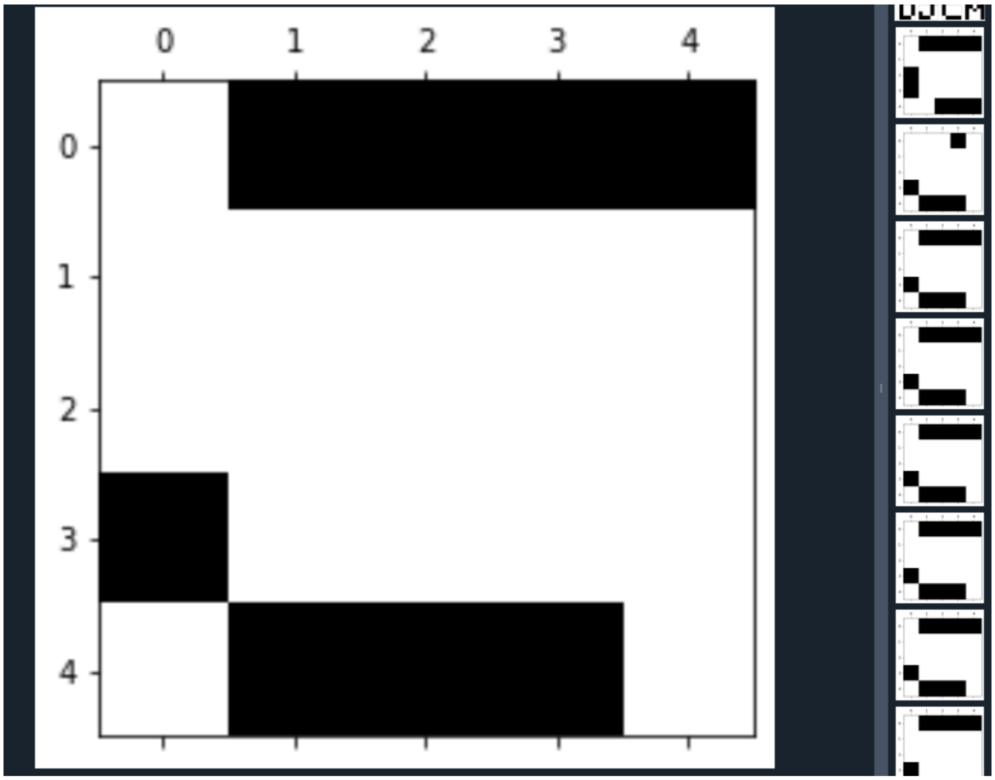
```
            w += patterns[n, i] * patterns[n, j]

        W[i, j] = w / patterns.shape[0]
        W[j, i] = W[i, j]
S = np.array([1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1.],
             dtype=np.float)

# Show the corrupted pattern
fig, ax = plt.subplots()
ax.matshow(S.reshape((pattern_height, pattern_width)), cmap='gray')
h = np.zeros((pattern_width * pattern_height))
# Defining Hamming Distance matrix for seeing convergence
hamming_distance = np.zeros((max_iterations, nb_patterns))
for iteration in range(max_iterations):
    for i in range(pattern_width * pattern_height):
        i = np.random.randint(pattern_width * pattern_height)
        h[i] = 0
        for j in range(pattern_width * pattern_height):
            h[i] += W[i, j]*S[j]
        S = np.where(h < 0, -1, 1)
    for i in range(nb_patterns):
        hamming_distance[iteration, i] = ((patterns - S)[i] != 0).sum()

    fig, ax = plt.subplots()
    ax.matshow(S.reshape((pattern_height, pattern_width)), cmap='gray')
plt.show()
hamming_distance
```

Output:-



Ques10 - Implement Self organizing maps for any given problem.

Code:-

```
import math


class SOM:

    # Function here computes the winning vector
    # by Euclidean distance
    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

    # Function here updates the winning vector
    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():

    # Training Examples ( m, n )
    T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]

    m, n = len(T), len(T[0])

    # weight initialization ( n, C )
    weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]

    # training
    ob = SOM()

    epochs = 3
    alpha = 0.5

    for i in range(epochs):
        for j in range(m):

            # training sample
            sample = T[j]
```

```
            # Compute winner vector
            J = ob.winner(weights, sample)

            # Update winning vector
            weights = ob.update(weights, sample, J, alpha)

    # classify test sample
    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, 0.8, 0.5, 0.9], [0.3333984375, 0.0666015625, 0.7, 0.3]]
```