SOFT COMPUTING

LAB FILE

Name : Aman Dhattarwal
Class: IT-1
Roll No: 707/IT/15

Q1) Using Neural Network Perform the implementation of AND, NAND, OR, NOR.

```
import numpy as np

def perceptron(weights, inputs, bias):
    model = np.add(np.dot(inputs, weights), bias)
    logit = activation_function(model, type="sigmoid")
    return np.round(logit)

def activation_function(model, type="sigmoid"):
    return {
        "sigmoid": 1 / (1 + np.exp(-model))
    }[type]

def compute(data, logic_gate, weights, bias):
    weights = np.array(weights)
    output = np.array([ perceptron(weights, datum, bias) for datum in data ])
    return output

def print_template(dataset, name, data):
    # act = name[6:]
    print("Logic Function: {}".format(name.upper()))
    print("X0\tX1\tX2\tY")
    toPrint = ["{1}\t{2}\t{3}\t{0}".format(output, *datas) for datas, output in zip(dataset, data)]
    for i in toPrint:
        print(i)

def main():
    dataset = np.array([
     [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
    gates = {
        "and": compute(dataset, "and", [1, 1, 1], -2),
        "or": compute(dataset, "or", [1, 1, 1], -0.9),
        "nand": compute(dataset, "nand", [-1, -1, -1], 3),
        "nor": compute(dataset, "nor", [-1, -1, -1], 1),
    }
    for gate in gates:
        print_template(dataset, gate, gates[gate])

main()
```

```
Logic Function: AND
X0      X1      X2      Y
0       0       0       0.0
0       0       1       0.0
0       1       0       0.0
0       1       1       0.0
1       0       0       0.0
1       0       1       0.0
1       1       0       0.0
1       1       1       1.0
Logic Function: OR
X0      X1      X2      Y
0       0       0       0.0
0       0       1       1.0
0       1       0       1.0
0       1       1       1.0
1       0       0       1.0
1       0       1       1.0
1       1       0       1.0
1       1       1       1.0
```

```
Logic Function: NAND
X0      X1      X2      Y
0       0       0       1.0
0       0       1       1.0
0       1       0       1.0
0       1       1       1.0
1       0       0       1.0
1       0       1       1.0
1       1       0       1.0
1       1       1       0.0
Logic Function: NOR
X0      X1      X2      Y
0       0       0       1.0
0       0       1       0.0
0       1       0       0.0
0       1       1       0.0
1       0       0       0.0
1       0       1       0.0
1       1       0       0.0
1       1       1       0.0
```

Q2) Using Neural Network Perform any two class classification problem

```
## Classify between 2 letters designed on a 3x3 matrix with classes 1 and -1
## Letters can be I and U

matrix_I = [1,1,1,-1,1,-1,1,1,1]
matrix_U =  [1,-1,1,1,-1,1,1,1,1]
target = [1,-1]
## Initialise the weights and bias as 0
weights_bias = [0] + [ 0 for i in range(9) ]

## Using Hebb's Rule to update weights
## for matrix_I
for i in range(10):
    if i==0:
        ## Update bias
        weights_bias[0]+=target[0]
    else:
        weights_bias[i]+=target[0]*matrix_I[i-1]
print "Bias and Weights after 1st training on I matrix:"
print weights_bias

## for matrix_U
for i in range(10):
    if i==0:
        ##Update bias
        weights_bias[0]+=target[1]
    else:
        weights_bias[i]+=target[1]*matrix_U[i-1]
print "Bias and Weights after 2nd training on U matrix:"
print weights_bias
```


```
## Classify between 2 letters designed on a 3x3 matrix with classes 1 and -1
## Letters can be I and U

matrix_I = [1,1,1,-1,1,-1,1,1,1]
matrix_U =  [1,-1,1,1,-1,1,1,1,1]
target = [1,-1]
## Initialise the weights and bias as 0
weights_bias = [0] + [ 0 for i in range(9) ]

## Using Hebb's Rule to update weights
## for matrix_I
for i in range(10):
    if i==0:
        ## Update bias
        weights_bias[0]+=target[0]
    else:
        weights_bias[i]+=target[0]*matrix_I[i-1]
```

```
print "Bias and Weights after 1st training on I matrix:"
print weights_bias

## for matrix_U
for i in range(10):
    if i==0:
        ##Update bias
        weights_bias[0]+=target[1]
    else:
        weights_bias[i]+=target[1]*matrix_U[i-1]
print "Bias and Weights after 2nd training on U matrix:"
print weights_bias
```

```
Bias and Weights after 1st training on I matrix:
[1, 1, 1, 1, -1, 1, -1, 1, 1, 1]
Bias and Weights after 2nd training on U matrix:
[0, 0, 2, 0, -2, 2, -2, 0, 0, 0]
```

Q3) Using Neural Network Perform any two class classification problem

```
## Classify between 2 letters designed on a 3x3 matrix with classes 1 and -1
## Letters can be H and L

matrix_H = [1,-1,1,1,1,1,1,-1,1]
matrix_L =  [1,-1,-1,1,-1,-1,1,1,1]
target = [1,-1]
## Initialise the weights and bias as 0
weights_bias = [0] + [ 0 for i in range(9) ]

## Using Hebb's Rule to update weights
## for matrix_I
for i in range(10):
    if i==0:
        ## Update bias
        weights_bias[0]+=target[0]
    else:
        weights_bias[i]+=target[0]*matrix_H[i-1]
print "Bias and Weights after 1st training on H matrix:"
print weights_bias

## for matrix_U
for i in range(10):
    if i==0:
        ##Update bias
        weights_bias[0]+=target[1]
    else:
        weights_bias[i]+=target[1]*matrix_L[i-1]
print "Bias and Weights after 2nd training on L matrix:"
print weights_bias
```

```
Bias and Weights after 1st training on H matrix:
[1, 1, -1, 1, 1, 1, 1, 1, -1, 1]
Bias and Weights after 2nd training on L matrix:
[0, 0, 0, 2, 0, 2, 2, 0, -2, 0]
```

Q4) Implement Perceptron Learning Algorithm

```
# Make a prediction with weights
def predict(row, weights):
    activation = weights[0]
    for i in range(len(row)-1):
        activation += weights[i + 1] * row[i]
    if activation > 0:
        return 1
    elif activation == 0:
        return 0
    else:
        return-1

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
    weights = [0 for i in range(len(train[0]))]
    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            prediction = predict(row, weights)
            if prediction != row[-1]:
                error = row[-1]
            else:
                error = 0
            sum_error += (row[-1] - prediction)**2
            weights[0] = weights[0] + l_rate * error
            for i in range(len(row)-1):
                weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
        print('epoch = %d error = %.3f'%(epoch + 1,sum_error))
    return weights

# Calculate weights
dataset = [[2.7810836,2.550537003,-1],
[1.465489372,2.362125076,-1],
[3.396561688,4.400293529,-1],
[1.38807019,1.850220317,-1],
[3.06407232,3.005305973,-1],
[7.627531214,2.759262235,1],
[5.332441248,2.088626775,1],
```

```
[6.922596716,1.77106367,1],
[8.675418651,-0.242068655,1],
[7.673756466,3.508563011,1]]
l_rate = 0.1
n_epoch = 4
print( 'lrate = %.3f, epochs = %d'% (l_rate, n_epoch))
weights = train_weights(dataset, l_rate, n_epoch)
print('weights =', weights)
```

```
lrate = 0.100, epochs = 4
epoch = 1 error = 5.000
epoch = 2 error = 4.000
epoch = 3 error = 0.000
epoch = 4 error = 0.000
weights = [-0.1, 0.20653640140000007, -0.23418117710000003]
```

Q5) Using Neural Network Implement XOR and XNOR gates

```
## For XOR Gate

## The Function: f1 = x1(~x2)
f1 = [ (0,0,0), (0,1,0), (1,0,1), (1,1,0) ]
## The Function: f2 = (~x1)x2
f2 = [ (0,0,0), (0,1,1), (1,0,0), (1,1,0) ]
## The Function: f3 = f1+f2
f3 = [ (0,0,0) ,(0,1,1), (1,0,1), (0,0,0)]


## For XNOR Gate
## Function: f4 = x1x2
f4 = [ (0,0,0), (0,1,0), (1,0,0), (1,1,1) ]
## Function: f5 = (~x1)(~x2)
f5 = [ (0,0,1), (0,1,0), (1,0,0), (1,1,0) ]
## Function: f6 = f4 + f5
f6 = [ (0,1,1), (0, 0, 0), (0, 0, 0), (1, 0, 1)]


def train(weights, gate):
    return [ weights[0]*gate[i][0] + weights[1]*gate[i][1] for i in range(len(gate)) ]

## We model the XOR function as a 2 layer neural network
## The first layer is computes function f1, the second layer computes function f2, finally, the
output is defined by ORing
## the outputs of the last layer.

def getthreshold(gate, value):
    l1 = []
    l2 = []
    for i in range(len(gate)):
        if gate[i][2]==0:
            l1.append(value[i])
        else:
            l2.append(value[i])

    if abs(max(l1)-min(l2))==1:
        return max(l1)
    else:
        return min(l1)

weights_f1 = [1,-1]
weights_f2 = [-1,1]
weights_f3 = [1, 1]
weights_f4 = [1, 1]
weights_f5 = [-1,-1]
weights_f6 = [1, 1]
```

```
vals_f1 = train(weights_f1, f1)
vals_f2 = train(weights_f2, f2)
vals_f3 = train(weights_f3, f3)
vals_f4 = train(weights_f4, f4)
vals_f5 = train(weights_f5, f5)
vals_f6 = train(weights_f6, f6)

print "----------------Modelling XOR Gate------------------"
print "Values for f1:", vals_f1
print "Values for f2:", vals_f2
print "Values for final output:", vals_f3

print "Threshold for first layer:", getthreshold(f1, vals_f1)
print "Threshold for second layer:", getthreshold(f2, vals_f2)
print "Threshold for output:", getthreshold(f3, vals_f3)

print "----------------Modelling XNOR Gate-------------------"
print "Values for f4:", vals_f4
print "Values for f5:", vals_f5
print "Values for final output:", vals_f6

print "Threshold for first layer:", getthreshold(f4, vals_f4)
print "Threshold for second layer:", getthreshold(f5, vals_f5)
print "Threshold for output", getthreshold(f6, vals_f6)
```

```
----------------Modelling XOR Gate------------------
Values for f1: [0, -1, 1, 0]
Values for f2: [0, 1, -1, 0]
Values for f3: [0, 1, 1, 0]
Threshold for first layer: 0
Threshold for second layer: 0
Threshold for output: 0
----------------Modelling XNOR Gate-------------------
Values for f4: [0, 1, 1, 2]
Values for f5: [0, -1, -1, -2]
Values for f6: [1, 0, 0, 1]
Threshold for first layer: 1
Threshold for second layer: -1
Threshold for output 0
```

Q6) Implement McCulloch-Pits Neural Network
//for AND gate

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import csv
import sys

class mcpitts:

    def __init__(self,filename,w1,w2,threshold):

        # read the input from excel file
        dataframe = filename

        # Convert the dataframe into lists for analysis
        self.feature1 = dataframe['x1'].tolist()
        self.feature2 = dataframe['x2'].tolist()
        self.actualop = dataframe['y'].tolist()

        # Set the weights and activation value
        self.w1 = w1
        self.w2 = w2
        self.threshold = threshold

    def calculate(self):

        # Obtain output by multiplying inputs and weights
        self.op = [ (self.feature1[i]*self.w1+self.feature2[i]*self.w2) for i in range(4) ]

        # Apply the activation function
        self.finalop = [ 1 if x >= self.threshold else 0 for x in self.op ]

        return self.finalop

    def display(self):
        print("Model outputs : ")
        print(self.finalop)

def main():
    w1 = 1
    w2 = 1
    threshold = 2
    data = [[0,0,0],[0,1,0],[1,0,0],[1,1,1]]
    filename = pd.DataFrame(data,columns=['x1','x2','y'])
    mc = mcpitts(filename,w1,w2,threshold)
    mc.calculate()
    mc.display()

main()
```

```
Model outputs :
[0, 0, 0, 1]
```

Q7) Implement Hebb Network

```
# import the required modules
import numpy as np
import pandas as pd

class hebb:

    def __init__(self,filename,threshold=0):

        # Set the threshold value
        self.threshold = threshold

        # read the input from excel file
        dataframe = filename

        # find out the number of features
        self.no_features = len(dataframe.columns) - 1
        # find out the number of inputs
        self.no_rows = len(dataframe.index)

        # Convert the dataframe into lists for analysis
        self.training_data = [ dataframe.iloc[i,:self.no_features].tolist() for i in
range(self.no_rows) ]
        # Obtain the output in a separate list
        self.actual_op = dataframe['y'].tolist()

        # Initialize the weights and bias as zero
        self.weights = [ 0 for x in range(self.no_features)]
        self.bias = 0


    def displayresults(self):
        # Display the results of the model
        print("The weights are : ")
        print(self.weights+[self.bias])
        print("---------------------")

    def calculate(self):

        # Learn weights using Hebb Model
        for x in range(self.no_rows):
```

```
            for y in range(self.no_features):
                self.weights[y] += self.training_data[x][y]*self.actual_op[x]
            self.bias += 1*self.actual_op[x]

        # Obtain output by multiplying inputs and weights
        yin = (np.sum((np.array(self.training_data)*np.array(self.weights)),axis=1)+np.array(self.bias)).tolist()

        # Apply the activation function
        self.finalop = [ 1 if x >= self.threshold else -1 for x in yin ]

        return self.finalop

def main():

    threshold = 0
    data = [[0,0,0],[0,1,0],[1,0,0],[1,1,1]]
    filename = pd.DataFrame(data,columns=['x1','x2','y'])
    hb = hebb(filename,threshold)
    hb.calculate()

main()
```

```
The weights are :
[1, 1, 1]
---------------------
```

Q8) Implement Adaline network.

```
# Make a prediction with weights
def predict(row, weights):
	activation = weights[0]
	for i in range(len(row)-1):
		activation += weights[i + 1] * row[i]
	return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
	weights = [0.01 for i in range(len(train[0]))]
	for epoch in range(n_epoch):
		sum_error = 0.0
		for row in train:
			prediction = predict(row, weights)
			error = row[-1] - prediction
			sum_error += error**2
			weights[0] = weights[0] + l_rate * error
			for i in range(len(row)-1):
				weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
		print('epoch=%d, error=%.3f' % (epoch, sum_error))
	return weights

# Calculate weights
dataset = [[2.7810836,2.550537003,0],		[1.465489372,2.362125076,0],
	[3.396561688,4.400293529,0],		[1.38807019,1.850220317,0],
	[3.06407232,3.005305973,0],	[7.627531214,2.759262235,1],
	[5.332441248,2.088626775,1],		[6.922596716,1.77106367,1],
	[8.675418651,-0.242068655,1],		[7.673756466,3.508563011,1]]
l_rate = 0.1
n_epoch = 5
print( 'lrate = %.3f'% (l_rate))
weights = train_weights(dataset, l_rate, n_epoch)
print('weights =', weights)
```

```
lrate = 0.100
epoch=0, error=2.000
epoch=1, error=1.000
epoch=2, error=0.000
epoch=3, error=0.000
epoch=4, error=0.000
weights = [-0.1, 0.20653640140000007, -0.23418117710000003]
```

Q9) Implement Heteroassociative Neural Network

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import csv
import sys

class heteroassociative:

    def __init__(self,filename,features,threshold):

        # store number of features
        self.num_features = features
        # store the threshold value
        self.threshold = threshold

        # read the input from excel file
        excel_file = filename
        # convert it into a pandas dataframe
        dataframe = pd.read_excel(excel_file)

        # Convert dataframe to numpy array
        self.wholeset = dataframe.values
        # Separate the inputs and outputs
        self.train_data = self.wholeset[:,:self.num_features]
        self.train_op = self.wholeset[:,self.num_features:]

    def train(self):
        # calculate the associative weights
        self.weights = (self.train_data.T)@(self.train_op)

    def test(self):
        # calculate the results
        self.op = self.train_data@self.weights

        # apply the activation function to the op
        self.op = self.op > self.threshold
        self.op = self.op.astype(int)

    def display(self):

        print("Weights : ")
        print(self.weights)

        print("Outputs : ")
        print(self.op)


if __name__ == '__main__':

    # path to the input file
    filename = sys.argv[1]
```

```
features = int(sys.argv[2])
threshold = float(sys.argv[3])

# Initialize the model
h = heteroassociative(filename,features,threshold)

h.train()
h.test()
h.display()
```