ARTIFICIAL NEURAL NETWORK

ITITE26

PRACTICAL FILE

Submitted by:-

ABHINAV SHARMA

2019UIT3119

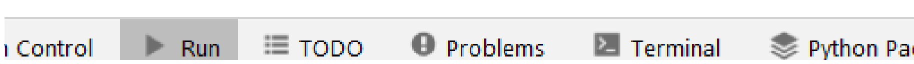
IT section 2

1. Realize MCP model for logic OR gate.

```
x1 = [0,0,1,1]
x2 = [0,1,0,1]
y = [0,1,1,1]
z = [0,0,0,0]
yin=0

w1=int(input("Enter weight w1: "))
w2=int(input("Enter weight w2: "))
theta=int(input("Enter threshold value: "))
for x in range(4):
    yin=w1*x1[x] + w2*x2[x]
    if(yin>=theta):
        z[x]=1
    else:
        z[x]=0

if(z==y):
    print("OR gate is recognized with given values")
else:
    print("Try again with different values")
```

output:-

```
Enter weight w1: 1
Enter weight w2: 1
Enter threshold value: 1
OR gate is recognized with given values

Process finished with exit code 0
```



2. Realize MCP model for logic XOR gate

```
x1 = [0,0,1,1]
x2 = [0,1,0,1]
z1 = [0,0,0,0]
z2 = [0,0,0,0]
y =  [0,1,1,0]
z =  [0,0,0,0]
yin1=0
yin2=0
yin=0

print("Enter Weights and threshold for hidden layer")
w11=int(input("Enter weight w11: "))
w21=int(input("Enter weight w21: "))
theta1=int(input("Enter threshold theta1: "))
w12=int(input("Enter weight w12: "))
w22=int(input("Enter weight w22 "))
theta2=int(input("Enter threshold theta2: "))


print("Enter weights for output layer")
w1=int(input("Enter weight w1: "))
w2=int(input("Enter weight w2: "))
theta=int(input("Enter threshold theta: "))

for x in range(4):

    yin1=w11*x1[x]+w21*x2[x]
    #print("yin1 :",yin1)
    if(yin1>=theta1):
        z1[x]=1
    else:
        z1[x]=0
print("z1: ",z1)

for x in range(4):

    yin2=(w12*x1[x])+(w22*x2[x])
    print("yin2 :", yin2)
 if(yin2>=theta2):
        z2[x]=1
    else:
```

```
        z2[x]=0
print("z2",z2)

for x in range(4):
    yin=w1*z1[x] + w2*z2[x]
    if(yin>=theta):
        z[x]=1
    else:
        z[x]=0

print("z: ",z)
if(z==y):
    print("XOR gate is recognized with given values")
else:
    print("Try again with different values")
```

```
Enter Weights and threshold for hidden layer
Enter weight w11: 1
Enter weight w21: -1
Enter threshold theta1: 1
Enter weight w12: -1
Enter weight w22 1
Enter threshold theta2: 1
Enter weights for output layer
Enter weight w1: 1
Enter weight w2: 1
Enter threshold theta: 1
z1:  [0, 0, 1, 0]
yin2 : 0
yin2 : 1
yin2 : -1
yin2 : 0
z2 [0, 1, 0, 0]
z:  [0, 1, 1, 0]
XOR gate is recognized with given values

Process finished with exit code 0
```

3. Implement Hebb learning rule for AND logic gate.

```
x1=[1,1,-1,-1]
x2=[1,-1,1,-1]
y=[1,-1,-1,-1]
w1=0
w2=0
b=0
epoch=int(input("Enter number of  epochs: "))
for i in range(epoch):
    for j in range(4):
        w1=w1+x1[j]*y[j]
        w2=w2+x2[j]*y[j]
        b=b+y[j]

print("Final weights are: ")
print("w1: ",w1)
print("w2: ",w2)
print("b: ",b)
```

```
Enter number of  epochs: 2
Final weights are:
w1:  4
w2:  4
b:  -4

Process finished with exit code 0
```

4. Implement Perceptron learning for AND NOT logic gate

```python
x1=[1,1,-1,-1]
x2=[1,-1,1,-1]
t=[-1,1,-1,-1]

w1=0
w2=0
b=0
yin=0
theta=float(input("Enter value of threshold: "))
alpha=int(input("Enter value of Learning rate alpha: "))
epoch=int(input("Enter the number of epochs: "))

def activation(yin):
    if(yin>theta):
        return 1
    elif(yin<-theta):
        return -1
    else:
        return 0

for j in range(epoch):
    for i in range(4):
        yin=w1*x1[i]+w2*x2[i]+b
        y=activation(yin)
        if(y!=t[i]):
            w1=w1+alpha*t[i]*x1[i]
            w2=w2+alpha*t[i]*x2[i]
            b=b+alpha*t[i]

print("Final Weights are: ")
print("w1: ",w1)
print("w2: ",w2)
print("b: ",b)
```

```
Enter value of threshold: 0
Enter value of Learning rate alpha: 1
Enter the number of epochs: 1
Final Weights are:
w1:  1
w2:  -1
b:  -1

Process finished with exit code 0
```

5. Use ADALINE network to implement OR logic gate.

```
x1=[1,1,-1,-1]
x2=[1,-1,1,-1]
t=[1,1,1,-1]

w1=0.1
w2=0.1
b=0.1
yin=0.0
E=0.0

alpha=float(input("Enter value of Learning rate: "))
epoch=int(input("Enter the number of epochs: "))


for j in range(epoch):
    E=0.0
    for i in range(4):
        yin=w1*x1[i]+w2*x2[i]+b
        w1=w1+alpha*(t[i]-yin)*x1[i]
        w2=w2+alpha*(t[i]-yin)*x2[i]
        b=b+alpha*(t[i]-yin)
        E=E+(t[i]-yin)**2
    print("Mean Square Error of epoch",j+1,": ","%.3f" %E)

print("Final Weights are: ")
print("w1: ","%.4f" %w1)
print("w2: ","%.4f" %w2)
print("b: ","%.4f" %b)
```

```
Enter value of Learning rate: 0.02
Enter the number of epochs: 5
Mean Square Error of epoch 1 :  2.941
Mean Square Error of epoch 2 :  2.652
Mean Square Error of epoch 3 :  2.407
Mean Square Error of epoch 4 :  2.199
Mean Square Error of epoch 5 :  2.023
Final Weights are:
w1:  0.2369
w2:  0.2398
b:  0.2341

Process finished with exit code 0
```

6. Implement Backpropagation algorithm to classify a dataset.

```python
# Backprop on the Seeds Dataset
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp


# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())


# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
```

```
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup


# Find the min and max values for each column
def dataset_minmax(dataset):
    minmax = list()
    stats = [[min(column), max(column)] for column in
zip(*dataset)]
    return stats


# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row) - 1):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] -
minmax[i][0])


# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split


# Calculate accuracy percentage
```

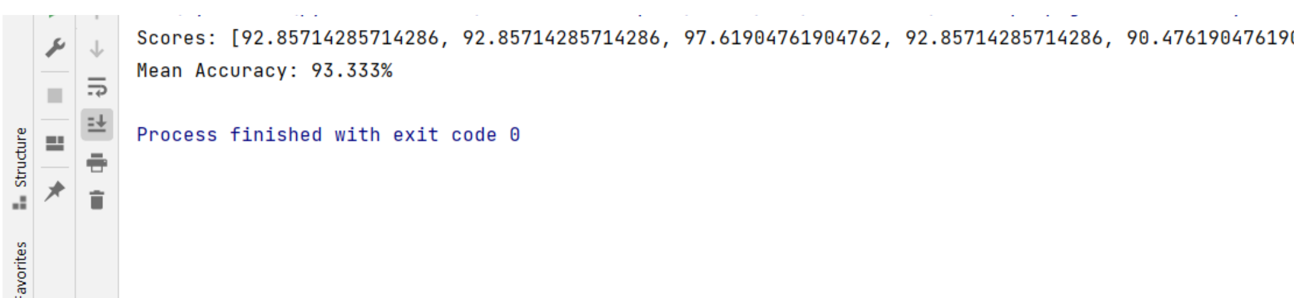
```
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0


# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores


# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights) - 1):
        activation += weights[i] * inputs[i]
    return activation
```

```
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))


# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs


# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)


# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network) - 1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
```

```
                neuron = layer[j]
                errors.append(neuron['output'] - expected[j])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] *
transfer_derivative(neuron['output'])


# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i -
1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] -= l_rate * neuron['delta'] *
inputs[j]
            neuron['weights'][-1] -= l_rate * neuron['delta']


# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch,
n_outputs):
    for epoch in range(n_epoch):
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)


# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
```

```
    network = list()
    hidden_layer = [{'weights': [random() for i in
range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights': [random() for i in
range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network


# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))


# Backpropagation Algorithm With Stochastic Gradient
Descent
def back_propagation(train, test, l_rate, n_epoch,
n_hidden):
    n_inputs = len(train[0]) - 1
    n_outputs = len(set([row[-1] for row in train]))
    network = initialize_network(n_inputs, n_hidden,
n_outputs)
    train_network(network, train, l_rate, n_epoch,
n_outputs)
    predictions = list()
    for row in test:
        prediction = predict(network, row)
        predictions.append(prediction)
    return (predictions)


# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
```

```
filename = 'wheat-seeds.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0]) - 1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0]) - 1)
# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
# evaluate algorithm
n_folds = 5
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation,
n_folds, l_rate, n_epoch, n_hidden)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores) /
float(len(scores))))
```



7. Implement Heteroassociative memory for pattern association

```
def activationFunction(value):
    if value > 0:
        return 1
    elif value == 0:
        return 0
    else:
        return -1


def trainingalgo(x, y, w, n, m):
    for i in range(n):
        for j in range(m):
            w[i][j] += x[i]*y[j]

    return w


def testingalgo(x, w, n, m):
    y = []
    for j in range(m):
        temp = 0
        for i in range(n):
            print(w[i][j])
            temp += x[i]*w[i][j]
        y.append(temp)
    return y


def main():
    print("--------* Training Begins *--------")
    rows = int(input("Enter number of inputs: "))
    n = int(input("Enter number of elements in input: "))
    m = int(input("Enter number of elements in output: "))
    w = []
    for i in range(n):
        temp = []
        for j in range(m):
```

```
            temp.append(0)
        w.append(temp)
    print(w)

    for i in range(rows):
        print("Enter input: ")
        x = input().strip().split(' ')
        x = list(int(a) for a in x)

        print("Enter output: ")
        y = input().strip().split(' ')
        y = list(int(a) for a in y)

        w = trainingalgo(x, y, w, n, m)
        print("Updated Weights :")

        for a in range(n):
            for b in range(m):
                print(w[a][b], end=" ")
            print("\n")

    print("--------* Testing Begins *-----")

    t = int(input("Enter number of testing inputs: "))

    for i in range(t):
        print("Enter input: ")
        x = input().strip().split(" ")
        x = list(int(a) for a in x)

        y_in = testingalgo(x, w, n, m)
        print("Output for current Input: ")
        for j in range(len(y_in)):
            print(activationFunction(y_in[j]), end=" ")
        print("\n")


main()
```

```
--------* Training Begins *--------
Enter number of inputs: 4
Enter number of elements in input: 4
Enter number of elements in output: 2
[[0, 0], [0, 0], [0, 0], [0, 0]]
Enter input:
1 0 1 0
Enter output:
1 0
Updated Weights :
1 0

0 0

1 0

0 0

Enter input:
1 0 0 1
Enter output:
1 0
Updated Weights :
2 0

0 0

1 0
```

```
1 0

Enter input:
1 1 0 0
Enter output:
0 1
Updated Weights :
2 1

0 1

1 0

1 0

Enter input:
0 0 1 1
Enter output:
0 1
Updated Weights :
2 1

0 1

1 1

1 1

--------* Testing Begins *-----
```

```
--------* Testing Begins *-----
Enter number of testing inputs: 1
Enter input:
0 1 1 1
2
0
1
1
1
1
1
1
Output for current Input:
1 1


Process finished with exit code 0
```

8. Implement Bidirectional Associative Memory for pattern recognition

```
# 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)

# 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))
```

```
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]]

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]]

Process finished with exit code 0
```

9. Implement hopfield network for any given problem.

```
import numpy as np

print("Input Vector: [1,1,1,-1]")
ipvector = np.array([1,1,1,-1]).reshape(1,4)

def activation(x):
    if(x>0):
        return 1
    else:
        return 0

w=np.transpose(ipvector)*ipvector
for i in range(4):
    for j in range(4):
        if(i==j):
            w[i][j]=0

print("\n Weight Matrix: ")
print(w)
ipvector = np.array([1,1,1,0])

print("Input vector in binary representation: ",ipvector)
x=np.array([0,0,1,0]).reshape(1,4)
y=x

for i in range(4):
    yin = x[0][i] + np.dot(y,w[:,i])
    print("yin_",i+1,": ",yin)
    y[0][i]=activation(yin)
    print("Updated Y :",y)
    if((y==ipvector).all()):
        print("Hence y=input vector x, vector converge")
        break
```

```
Input Vector: [1,1,1,-1]

 Weight Matrix:
[[ 0  1  1 -1]
 [ 1  0  1 -1]
 [ 1  1  0 -1]
 [-1 -1 -1  0]]
Input vector in binary representation:  [1 1 1 0]
yin_ 1 :  [1]
Updated Y : [[1 0 1 0]]
yin_ 2 :  [2]
Updated Y : [[1 1 1 0]]
Hence y=input vector x, vector converg

Process finished with exit code 0
```

10. Implement self-organizing maps

```
import numpy as np

ip = np.array([[0,0,1,1],
               [1,0,0,0],
               [0,1,1,0],
               [0,0,0,1]])

print("No of inputs:",4)
print("No of Cluster:",2)
print("Learning rate: ",0.5)
alpha=0.5
def Euclidian_distance(x,w):
    ans=0
    for i in range(4):
        ans=ans+(w[i]-x[i])**2
    return ans

def update(w,ip):
    for i in range(4):
        w[i] = w[i] +0.5*(ip[i]-w[i])

w= np.array([[0.2,0.4,0.6,0.8],
             [0.9,0.9,0.5,0.3]])

print("Initial Weights")
print(w)


for i in range(4):
    d1=Euclidian_distance(ip[i],w[0])
    d2=Euclidian_distance(ip[i],w[1])
    if(d1<d2):
        update(w[0],ip[i])
    else:
        update(w[0],ip[1])
    print("Updated Weights after input",i+1,)
    print(w)
```

```
No of inputs: 4
No of Cluster: 2
Learning rate:  0.5
Initial Weights
[[0.2 0.4 0.6 0.8]
 [0.9 0.9 0.5 0.3]]
Updated Weights after input 1
[[0.1 0.2 0.8 0.9]
 [0.9 0.9 0.5 0.3]]
Updated Weights after input 2
[[0.55 0.1  0.4  0.45]
 [0.9  0.9  0.5  0.3 ]]
Updated Weights after input 3
[[0.775 0.05  0.2   0.225]
 [0.9   0.9   0.5   0.3  ]]
Updated Weights after input 4
[[0.3875 0.025  0.1    0.6125]
 [0.9    0.9    0.5    0.3   ]]

Process finished with exit code 0
```