Batch: A2 **Roll No.: 1911027**

Experiment No. : 03

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Logic gates with multilayer neural network.

Objective: To Solve logic gates with two nodes in the hidden layer.

Expected Outcome of Experiment:

CO1 : Identify and describe soft computing techniques and their roles

Books/ Journals/ Websites referred:

- J.S.R.Jang, C.T.Sun and E.Mizutani, "Neuro-Fuzzy and Soft Computing", PHI, 2004, Pearson Education 2004.
- Davis E.Goldberg, "Genetic Algorithms: Search, Optimization and Machine Learning", Addison Wesley, N.Y., 1989.
- S. Rajasekaran and G.A.V.Pai, "Neural Networks, Fuzzy Logic and Genetic Algorithms", PHI, 2003.
- http://library.thinkquest.org/C007395/tqweb/history.html.

Pre Lab/ Prior Concepts:

Neural networks, sometimes referred to as connectionist models, are parallel-distributed models that have several distinguishing features-

1) A set of processing units;
2) An activation state for each unit, which is equivalent to the output of the unit;
3) Connections between the units. Generally each connection is defined by a weight w_{jk} that

determines the effect that the signal of unit *j* has on unit *k;*

4) A propagation rule, which determines the effective input of the unit from its external inputs;

5) An activation function, which determines the new level of activation based on the effective input and the current activation;

6) An external input (bias, offset) for each unit;

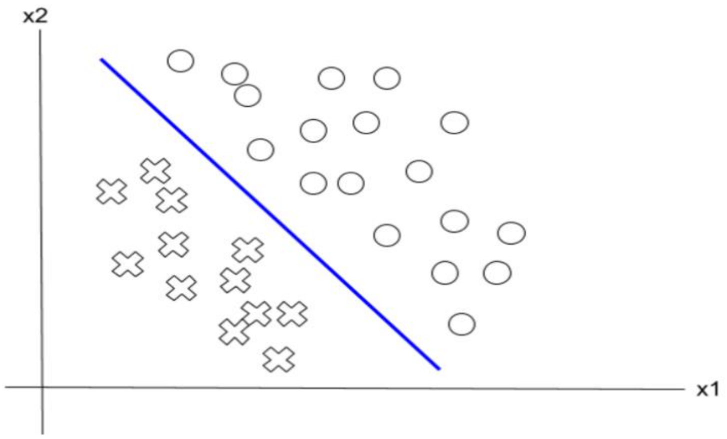
7) A method for information gathering (learning rule);

8) An environment within which the system can operate, provide input signals and, if necessary, error signals.

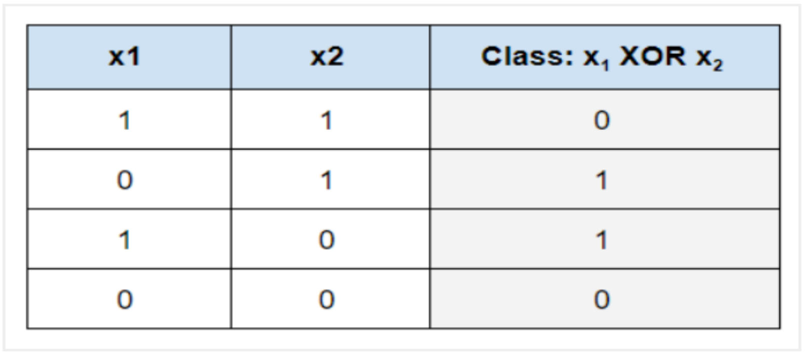
Implementation Details:

XOR problem :

Consider a data set with two attributes x1 and x2 and two classes 0 and 1. Let class 0 = o and class 1 = x.

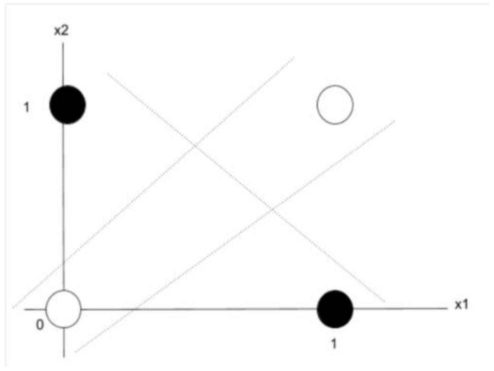


A straight line (or plane) can be used to separate the two classes (i.e. the x's from the o's). In other words, a single decision surface can be used as the boundary between both classes. When the data is nonlinearly separable, we cannot use linear models. We must instead use models that can handle nonlinearly separable data. Here is an example. Consider the XOR function acting on training data that has two attributes x1 and x2 and a class (0 or 1). The function x1 XOR x2 has the following truth table:



x1	x2	Class: x_1 XOR x_2
1	1	0
0	1	1
1	0	1
0	0	0

Now let us graph this function. For the class, black circles = 1 and white circles = 0.



We could try all day to try to place a line on the graph that can separate the classes, and we still would not succeed. This problem is known as the XOR problem and is a clear example of when linear classifiers like Naive Bayes and Logistic Regression would not work, and we would need to consider other types of machine learning models that can handle nonlinearly separable data.

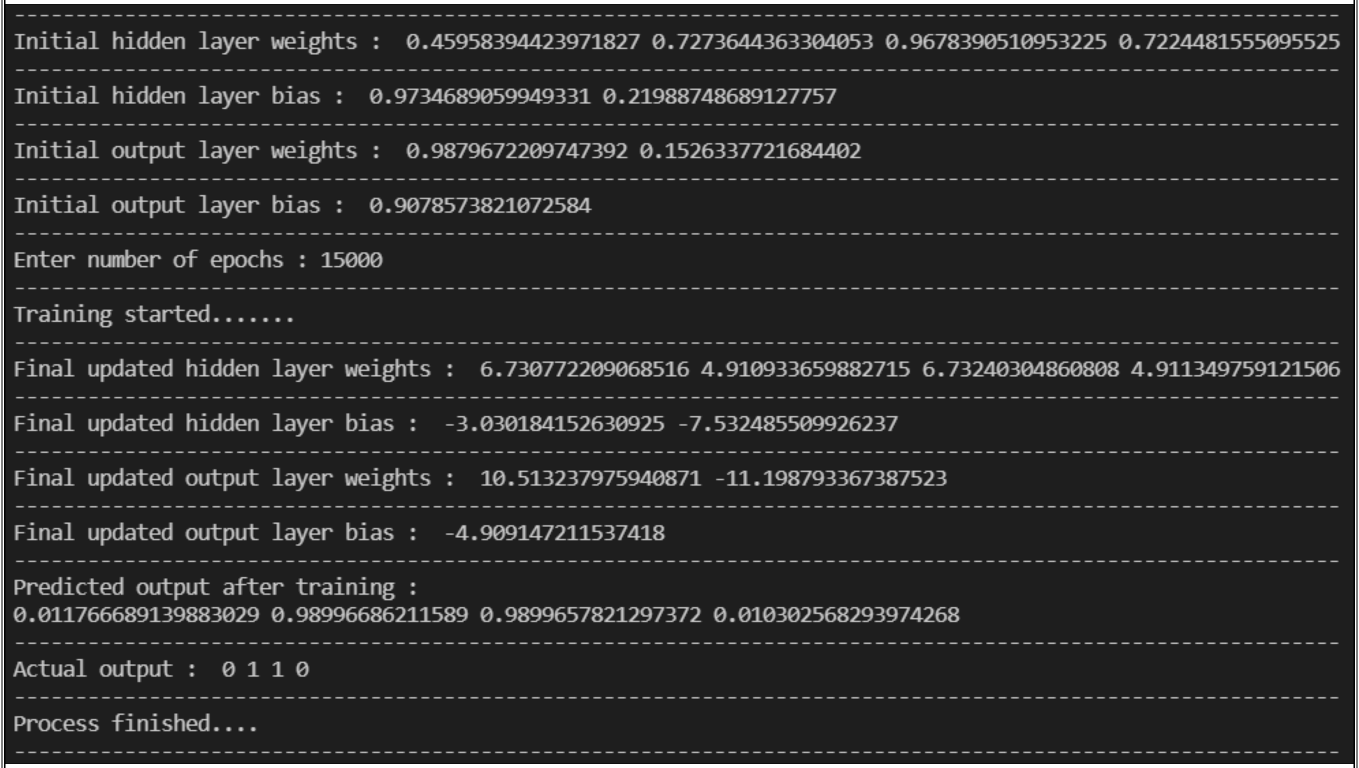
Algorithm :

1. Initialize the weights and biases randomly and learning factor as well.
2. Initialize input and output vector for XOR function.
3. Take number of epochs(from user or any predefined value).
4. Iterate till the number of epochs.
5. Calculate the predicted output using sigmoid function(in our case).
6. Computer error/loss.
7. Update the initial weights and biases.(weight=weight – delw* learning_rate and bias=bias - delb *learning_rate).
8. Repeat until number of epochs.

Code :

```
import numpy as np
def print_all(a,b,c,d):
    print("------------------------------------------------------------------------
--------------------------------")
    print("Initial hidden layer weights : ",end=" ")
    for i in a:
        for j in i:
            print(j,end=" ")
    print("\n----------------------------------------------------------------------
----------------------------------")
    print("Initial hidden layer bias : ",end=" ")
    for i in b:
        for j in i:
            print(j,end=" ")
    print("\n----------------------------------------------------------------------
----------------------------------")
    print("Initial output layer weights : ",end=" ")
    for i in c:
        for j in i:
            print(j,end=" ")
    print("\n----------------------------------------------------------------------
----------------------------------")
    print("Initial output layer bias : ",end=" ")
    for i in d:
        for j in i:
            print(j,end=" ")
    print("\n----------------------------------------------------------------------
----------------------------------")

def intermediate_cal(x,w_hidden_layer):
    activation_hidden=np.dot(x,w_hidden_layer)
    activation_hidden=activation_hidden+b_hidden_layer
    return activation_hidden

def intermediate_cal2(output_from_hidden,w_output_layer):
    output_layer_activation=np.dot(output_from_hidden,w_output_layer)
    output_layer_activation=output_layer_activation+b_output_layer
    return output_layer_activation

def processing(eps,w_hidden_layer,b_hidden_layer,w_output_layer,b_output_layer):
    for i in range(0,eps):
        activation_hidden=intermediate_cal(x,w_hidden_layer)
        output_from_hidden=sigmoid(activation_hidden)
        output_layer_activation=intermediate_cal2(output_from_hidden,w_output_layer)
        output_generated=sigmoid(output_layer_activation)
        error_generated=y-output_generated
        target_output=error_generated*sign_der(output_generated)
        error_hidden=target_output.dot(w_output_layer.T)
        target_hidden=error_hidden*sign_der(output_from_hidden)
        w_output_layer+=output_from_hidden.T.dot(target_output) * learning_rate
        b_output_layer+=np.sum(target_output,axis=0,keepdims=True) * learning_rate
        w_hidden_layer+=x.T.dot(target_hidden) * learning_rate
        b_hidden_layer+=np.sum(target_hidden,axis=0,keepdims=True) * learning_rate
    print_final(w_hidden_layer,b_hidden_layer,w_output_layer,b_output_layer,output_gene
rated)

def print_final(a,b,c,d,e):
    print("Final updated hidden layer weights : ",end=" ")
    for i in a:
        for j in i:
            print(j,end=" ")
    print("\n--------------------------------------------------------------------------
----------------------------------")
    print("Final updated hidden layer bias : ",end=" ")
    for i in b:
        for j in i:
            print(j,end=" ")
    print("\n--------------------------------------------------------------------------
----------------------------------")
    print("Final updated output layer weights : ",end=" ")
    for i in c:
        for j in i:
            print(j,end=" ")
    print("\n--------------------------------------------------------------------------
----------------------------------")
    print("Final updated output layer bias : ",end=" ")
    for i in d:
        for j in i:
            print(j,end=" ")
    print("\n--------------------------------------------------------------------------
----------------------------------")
    print("Predicted output after training : ")
    for i in e:
        for j in i:
            print(j,end=" ")
    print("\n--------------------------------------------------------------------------
----------------------------------")

def initializations(a,b):
    return np.random.uniform(size=(a,b))

def sigmoid(x):
    temp=(1+np.exp(-x))
    return 1/temp

def sign_der(x):
    return x*(1-x)

x=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([[0],[1],[1],[0]])
learning_rate=0.8
w_hidden_layer=initializations(2,2)
b_hidden_layer=initializations(1,2)
w_output_layer=initializations(2,1)
b_output_layer=initializations(1,1)
print_all(w_hidden_layer,b_hidden_layer,w_output_layer,b_output_layer)
epochs=int(input("Enter number of epochs : "))
print("--------------------------------------------------------------------------------
----------------------------")
print("Training started.......")
print("--------------------------------------------------------------------------------
----------------------------")
processing(epochs,w_hidden_layer,b_hidden_layer,w_output_layer,b_output_layer)
print("Actual output : ",end=" ")
for i in y:
    for j in i:
        print(j,end=" ")
print("\n------------------------------------------------------------------------------
------------------------------")
print("Process finished....")
print("--------------------------------------------------------------------------------
----------------------------")
```

Output :



```
------------------------------------------------------------------------------------------
Initial hidden layer weights :  0.45958394423971827 0.7273644363304053 0.9678390510953225 0.7224481555095525
------------------------------------------------------------------------------------------
Initial hidden layer bias :  0.9734689059949331 0.21988748689127757
------------------------------------------------------------------------------------------
Initial output layer weights :  0.9879672209747392 0.1526337721684402
------------------------------------------------------------------------------------------
Initial output layer bias :  0.9078573821072584
------------------------------------------------------------------------------------------
Enter number of epochs : 15000
------------------------------------------------------------------------------------------
Training started.......
------------------------------------------------------------------------------------------
Final updated hidden layer weights :  6.730772209068516 4.910933659882715 6.73240304860808 4.911349759121506
------------------------------------------------------------------------------------------
Final updated hidden layer bias :  -3.030184152630925 -7.532485509926237
------------------------------------------------------------------------------------------
Final updated output layer weights :  10.513237975940871 -11.198793367387523
------------------------------------------------------------------------------------------
Final updated output layer bias :  -4.909147211537418
------------------------------------------------------------------------------------------
Predicted output after training :
0.011766689139883029 0.98996686211589 0.9899657821297372 0.010302568293974268
------------------------------------------------------------------------------------------
Actual output :  0 1 1 0
------------------------------------------------------------------------------------------
Process finished....
------------------------------------------------------------------------------------------
```

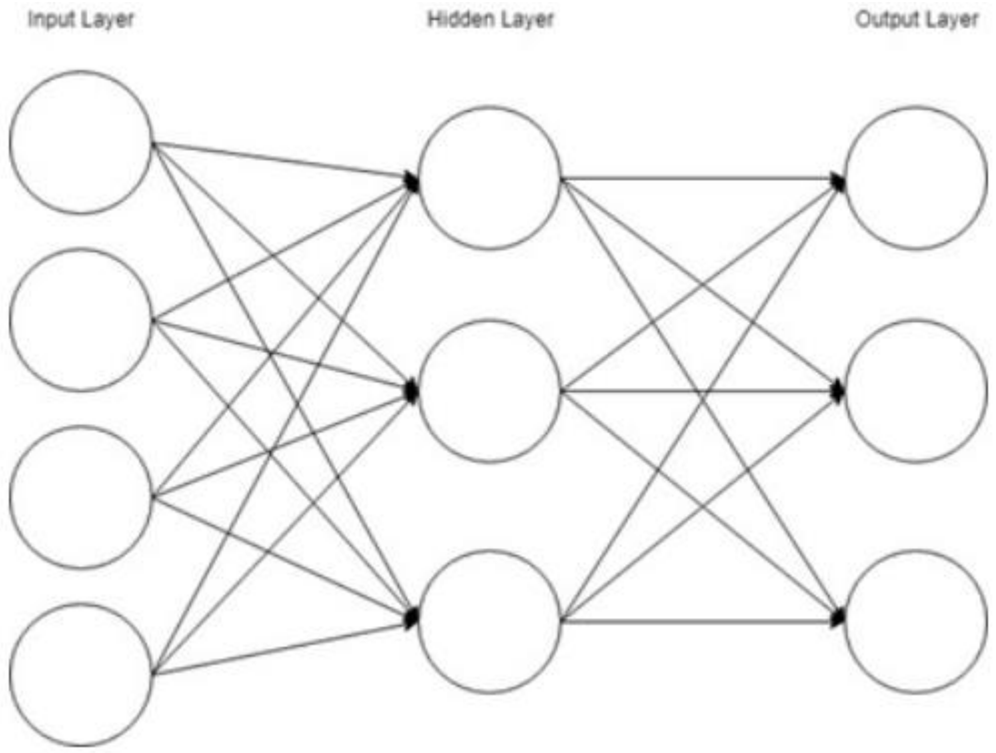
Conclusion: Thus, we have successfully implemented a logic gate with one hidden layer of Neural Network. As discussed in theory XOR problem is not linearly separable so we have to use multilayer perceptron with one hidden layer. Implemented weight updations in XOR problem using python programming language.

Post Lab Descriptive Questions :

1. Explain MLP.

ANS) A multilayer perceptron (MLP) is a feedforward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input and output layers. MLP uses backpropogation for training the network. MLP is a deep learning method. A multilayer perceptron is a neural network connecting multiple layers in a directed graph, which means that the signal path through the nodes only goes one way. Each node, apart from the input nodes, has a nonlinear activation function. An MLP uses backpropagation as a supervised learning technique. Since there are multiple layers of neurons, MLP is a deep learning technique. MLP is widely used for solving problems that require supervised learning as well as research into computational neuroscience and parallel distributed processing.

Applications include speech recognition, image recognition and machine translation. The Perceptron consists of an input layer and an output layer which are fully connected. MLPs have the same input and output layers but may have multiple hidden layers in between the aforementioned layers, as seen below.



The algorithm for the MLP is as follows:

1. Just as with the perceptron, the inputs are pushed forward through the MLP by taking the dot product of the input with the weights that exist between the input layer and the hidden layer (W---H). This dot product yields a value at the hidden layer. We do not push this value forward as we would with a perceptron though.
2. MLPs utilize activation functions at each of their calculated layers. There are many activation functions to discuss: rectified linear units (ReLU), sigmoid function, tanh. Push the calculated output at the current layer through any of these activation functions.
3. Once the calculated output at the hidden layer has been pushed through the activation function, push it to the next layer in the MLP by taking the dot product with the corresponding weights.
4. Repeat steps two and three until the output layer is reached.
5. At the output layer, the calculations will either be used for a backpropagation algorithm that corresponds to the activation function that was selected for the MLP (in the case of training) or a decision will be made based on the output (in the case of testing).

MLPs form the basis for all neural networks and have greatly improved the power of computers when applied to classification and regression problems. Computers are no longer limited by XOR cases and can learn rich and complex models thanks to the multilayer perceptron.

2. State any other linear inseperable problem.

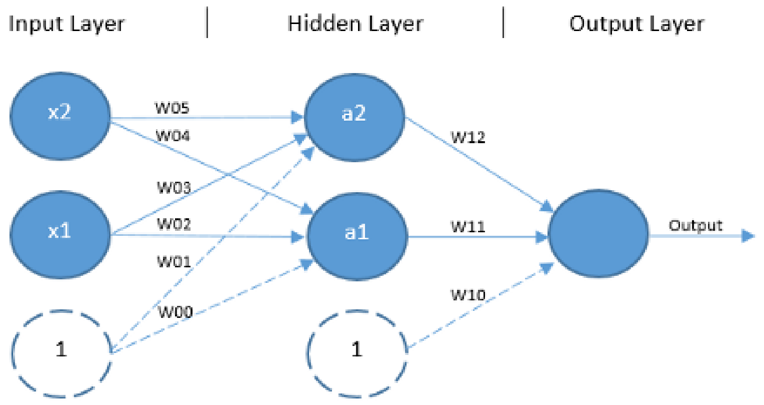
ANS) If the 2 sets of points cannot be separated by a line such that all the points of first set lies of one side of the line and another sets lies on another side of the line. Then the sets can be called as linearly inseperable as they cannot be separated by a single line. The another linearly inseperable problem can be EXNOR. If we consider the truth table of EXNOR

X1	X2	Y
0	0	1
0	1	0
1	0	0
1	1	1

When we plot this points on a graph it can be easily observed that there is no possibility that a single line can divide 2 classes (one for 0 and another for 1) and linear separability cannot be there and hence the EXNOR problem is linearly inseperable.

3. Explain why multilayer neural network is needed for linear inseparable problems.

ANS) A multilayer perceptron (MLP) is a feedforward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input and output layers. MLP uses backpropogation for training the network. MLP is a deep learning method. Multilayer networks solve the classification problem for non linear sets by employing hidden layers, whose neurons are not directly connected to the output. The additional hidden layers can be interpreted geometrically as additional hyper-planes, which enhance the separation capacity of the network. This new architecture introduces a new question: how to train the hidden units for which the desired output is not known. The Backpropagation algorithm offers a solution to this problem. A limitation of single perceptron architecture is that it is only capable of separating data points with a single line. This is particularly visible if you plot the XOR input values to a graph. The solution to this problem is to expand beyond the single-layer architecture by adding an additional layer of units without any direct access to the outside world, known as a hidden layer.



The products of the input layer values and their respective weights are parsed as input to the non-bias units in the hidden layer. Each non-bias hidden unit invokes an activation function — usually the classic sigmoid function in the case of the XOR problem — to squash the sum of their input values down to a value that falls between 0 and 1 (usually a value very close to either 0 or 1). The outputs of each hidden layer unit, including the bias unit, are then multiplied by another set of respective weights and parsed to an output unit. MLP architecture, while more complex than that of the classic perceptron network, is capable of achieving non-linear separation. MLP is widely used for solving problems that require supervised learning as well as research into computational neuroscience and parallel distributed processing. Applications include speech recognition, image recognition and machine translation. The processing unit of a single-layer perceptron network is able to categorize a set of patterns into two classes as the linear threshold function defines their linear separability. Conversely, the two classes must be linearly separable in order for the perceptron network to function correctly. This is the main limitation of a single-layer perceptron network.

1 layer perceptron ≡ 1 dimensional separator
2 layer perceptron ≡ 2 dimensional separator
3 layer perceptron ≡ 3 dimensional separator
n layer perceptron ≡ n dimensional separator

Since, linear inseparable problems outputs cannot be separated using a straight line, they cannot be classified using a single layer perceptron.

An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the nodes in the input layer, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable. Hence, a multilayer neural network is needed for linear inseparable problems.

Date: 16 / 9 / 2021 **Signature of faculty in-charge**