Batch: B2 **Roll No.: 16010121194**

Experiment No. 02

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

Signature of the Staff In-charge with date

Title: Implementation of XOR Gate LOGIC using perceptron network. .

Objective: To implement classifier using Multi-layer perceptron network for XOR logic of 2 inputs.

Expected Outcome of Experiment:

CO2 : Analyze various neural network architectures

Books/ Journals/ Websites referred:

- .

Pre Lab/ Prior Concepts:

Perceptron Model

The perceptron is a fundamental concept in machine learning and neural networks. It is one of the simplest artificial neural network architectures and serves as a building block for more complex models. The perceptron is a type of binary classifier that can make decisions based on input data.

Here's a basic overview of the perceptron model:

1. Inputs: The perceptron takes a set of input features (usually represented as a vector) and assigns a weight to each input feature. These weights represent the importance of each feature in making a decision.

2. Weights and Bias: In addition to the weights assigned to input features, the perceptron includes a bias term. The bias acts as an offset, allowing the model to shift the decision boundary.

3. Activation Function: The weighted sum of inputs and bias is then passed through an activation function. The purpose of the activation function is to introduce non-linearity into the model. The most commonly used activation function for the perceptron is the step function or its variants like the sigmoid function.

4. Output: The result of the activation function (often binary) determines the output of the perceptron. For example, in a binary classification task, the output could be 0 or 1, indicating which class the input belongs to.

5. Learning Algorithm: The perceptron learns from data using a learning algorithm that adjusts the weights and bias based on the error between the predicted output and the true output. One such algorithm is the perceptron learning rule, which updates weights and bias to minimize the classification error.

6. Training: During the training process, the perceptron iteratively adjusts its weights and bias using the learning algorithm, aiming to improve its ability to correctly classify examples from the training dataset.

It's important to note that a single perceptron is limited in its ability to solve complex problems, especially those that are not linearly separable. However, the concept of the perceptron forms the foundation for more advanced neural network architectures, such as multi-layer perceptrons (MLPs) and deep neural networks (DNNs), which can learn complex patterns and relationships in data.

In summary, the perceptron model is a basic building block of neural networks, used for binary classification tasks by assigning weights to input features, applying an activation function, and making decisions based on the resulting output.

Linear separability

Linear separability is a concept in machine learning and pattern recognition that refers to the ability to separate data points from different classes or categories using a straight line (in two-dimensional space), a hyperplane (in higher-dimensional space), or a linear decision boundary in general. Linear separability is closely associated with the notion of whether classes can be cleanly divided by a linear classifier.

In more intuitive terms, two classes of data points are said to be linearly separable if you can draw a straight line (or a hyperplane) in the feature space such that all data points from one class lie on one side of the line/hyperplane, and all data points from the other class lie on the other side. In other words, there's a clear "gap" between the two classes that a linear decision boundary can exploit for classification.

Linear separability is a fundamental concept in many machine learning algorithms, particularly in the context of binary classification. Many basic algorithms, such as the perceptron, support vector machines (SVMs), and logistic regression, work well when the data is linearly separable. However, real-world data often contains more complex patterns that cannot be separated by a straight line or hyperplane, necessitating the use of more advanced techniques like kernel methods and non-linear classifiers.

When data is not linearly separable, it may require transforming the feature space or using more complex models to capture the underlying patterns. Techniques such as kernel tricks and deep neural networks allow for the representation of non-linear decision boundaries, enabling the classification of data that is not linearly separable in the original feature space.

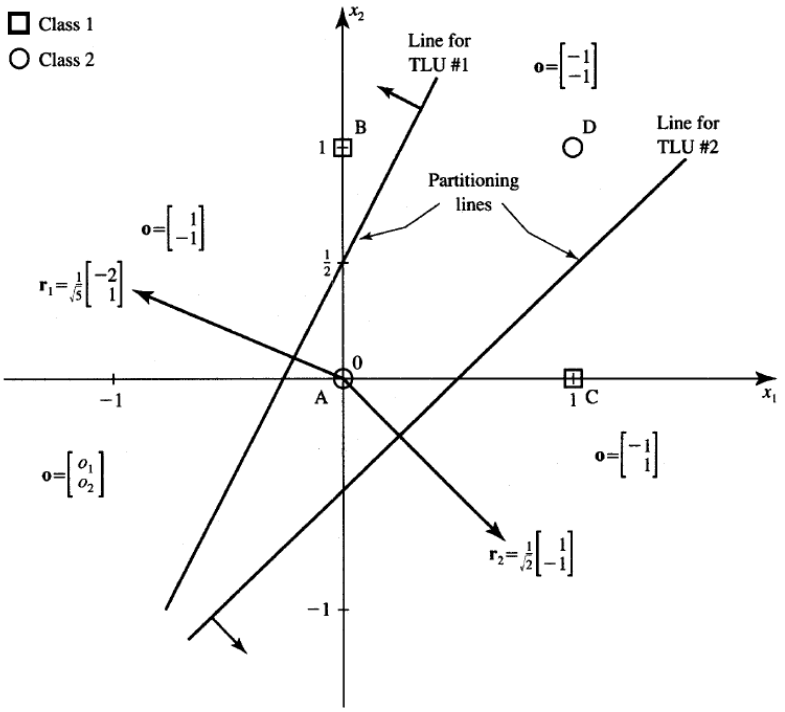
Design of Classifier using Multi-layer Perceptron model for XOR Gate logic with 2 inputs (Ref, Zurada 4.1 page no `68)

Truth table for XOR logic

	x_1	x_2	Output
	0	0	1
	0	1	−1
	1	0	−1
	1	1	1

(Ref, Zurada Chapter 4)

Perceptron (TLU #1 and TLU +2) for first layer using bipolar activation function



Two decision lines having equations using arbitrary selected partitioning as shown in above figure.

(Ref, Zurada Chapter 4)

$$-2x_1 + x_2 - \frac{1}{2} = 0$$

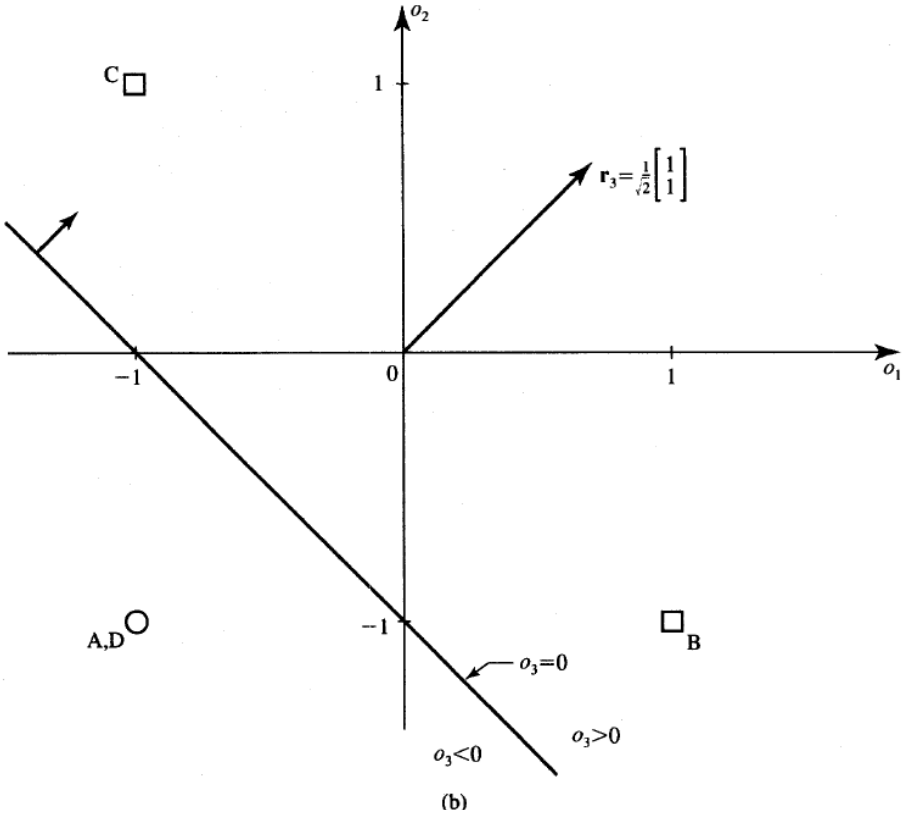
$$x_1 - x_2 - \frac{1}{2} = 0$$

Mapping performed by first layer perceptron's TLU#1 and TLU#2

$$o_1 = \text{sgn}\left(-2x_1 + x_2 - \frac{1}{2}\right)$$

$$o_2 = \text{sgn}\left(x_1 - x_2 - \frac{1}{2}\right)$$

Final Output layer perceptron TLU#3



(b)

(Ref, Zurada Chapter 4)

The decision line represents the equation

$$o_1 + o_2 + 1 = 0$$

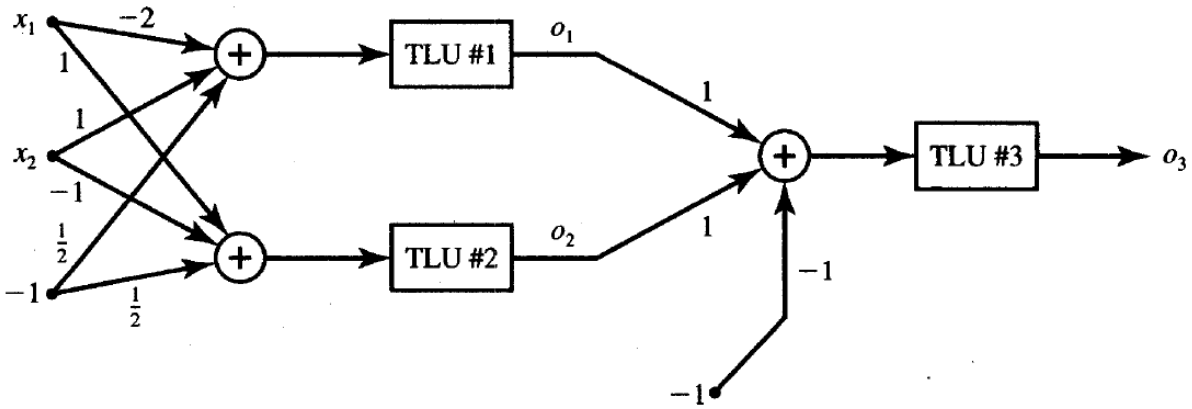
Mapping performed by output perceptron

The TLU #3 implements the decision o_3 as

$$o_3 = \text{sgn}(o_1 + o_2 + 1)$$

Classification summary table

Symbol	Pattern Space x_1	Pattern Space x_2	Image Space o_1	Image Space o_2	TLU #3 Input $o_1 + o_2 + 1$	Output Space o_3	Class Number
A	0	0	−1	−1	−	−1	2
B	0	1	1	−1	+	+1	1
C	1	0	−1	1	+	+1	1
D	1	1	−1	−1	−	−1	2



Classifier using Multi-layer Perception Network for implementing XOR logic for 2 inputs.

(Ref, Zurada Chapter 4)

Code:

```
#this code is written for the XOR gate whose diagram is given in the writeup.
#the values of weights are taken from the writeup background

#list of inputs to both neurons
inputs = [
    [1, 1],
    [0, 0],
    [1, 0],
    [0, 1]
]

class Perceptron():
```

```
  #declare the arrays and variables
  def __init__(self, inputs):
    self.inputs = inputs
    self.intermediate_output = []
    self.final_output = []
    self.class_classified = 0

  def calculating_output(self):

    #declaring variables res1 and res2 for storing result
    #declaring variables val1 and val2 for storing output before threshold is
applied
    res1 = 0
    res2 = 0
    val1 = 0
    val2 = 0

    #adding the product input given to the neurons with the weight of the neurons
    #along with product of bias input and weight
    res1 = self.inputs[0]*(-2) + self.inputs[1]*(1) - 1 * 1/2
    res2 = self.inputs[0]*(1) + self.inputs[1]*(-1) - 1 * 1/2

    #thresholding the intermediate outputs to obtain binary final outputs
    #val1 and val2 are the final outputs after applying threshold
    if(res1 >= 0):
      val1 = 1

    else:
      val1 = -1

    #if result is negative the output is -1
    #if result is position the output is +1

    if(res2 >= 0):
      val2 = 1

    else:
      val2 = -1

    #finding the value of val1 + val2 + 1 for classifying the output in either
Class 1 or Class 2
```

```
    #the weight and input of the bias is -1
    if val1 + val2 + (-1)*(-1) == -1:
      self.class_classified = 2

    elif val1 + val2 + (-1)*(-1) == 1:
      self.class_classified = 1

    #to store the output of both the neurons before the threshold was applied
    self.intermediate_output = [val1, val2]

    #inserting the value of val1 and val2 (outputs before threshold was applied)
to the array final_output[] for printing
    self.final_output = val1 + val2 + 1



print("Input","     Intermediate Output","      Output","     Final Class")

print()

#loop processes each input pattern and displays the intermediate outputs, final
output, and classified class
for i in inputs:

  #making an instance of the class Perceptron
  model = Perceptron(i)

  #function calculating_output() will be called for computing values
  model.calculating_output()
  print(i, "       ", model.intermediate_output, "               ",
model.final_output, "          ", model.class_classified)
  print()
```

Output:

```
PS C:\Users\uditi> & C:/Users/uditi/AppData/Local/Programs/Pyth
 Input      Intermediate Output       Output      Final Class

 [1, 1]         [-1, -1]               -1           2

 [0, 0]         [-1, -1]               -1           2

 [1, 0]         [-1, 1]                 1           1

 [0, 1]         [1, -1]                 1           1
```

Conclusion:

In this lab, I learned about the single layer and multilayer Perceptron model and implemented the XOR gate using the weights given in the write-up for the perceptron model in Python. I also learned how to classify outputs of the XOR gate. Concepts like linear separability were also understood.

Post Lab Descriptive Questions:

1. **Why is XOR-logic Gate function problem being Linearly non-separable?**

The XOR (exclusive OR) logic gate problem is considered linearly non-separable because it cannot be accurately represented or separated by a single linear decision boundary. In other words, you cannot draw a single straight line to separate the two classes of inputs and outputs in the XOR truth table.

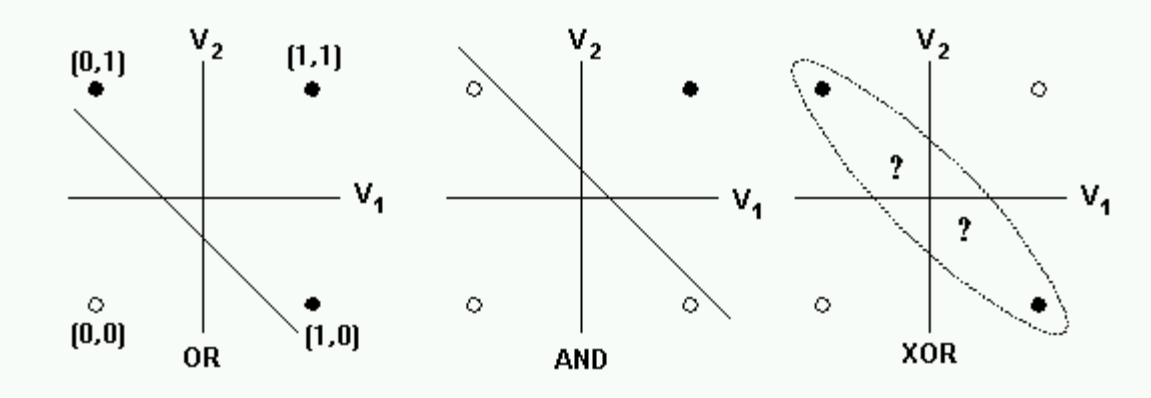
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In the XOR problem, the inputs (A and B) and the output (the result) can be either 0 or 1. When we plot these points on a two-dimensional plane, where the x-axis represents Input A and the y-axis represents Input B, we'll see that the points with Output 0 form one cluster (0,0) and (1,1), while the points with Output 1 form another cluster (0,1) and (1,0).

Because these two clusters are diagonal to each other, it's not possible to draw a single straight line that can accurately separate them. Linear classifiers like the perceptron or linear regression

can only draw straight lines, and they can't properly separate the XOR data. You need a non-linear decision boundary to effectively separate the XOR data points.

This is why the XOR problem is often used as an example to demonstrate the limitations of simple linear models and the need for more complex architectures like neural networks. Neural networks, particularly those with hidden layers and activation functions, can learn non-linear mappings and solve problems like XOR by creating multiple decision boundaries through their layers.



2. Design a classifier using MLP perceptron model for implementing NOT XOR logic

Code:

```
#List of inputs to both neurons
inputs = [
    [1, 1],
    [0, 0],
    [1, 0],
    [0, 1]
]

class Perceptron():

  #declare the arrays and variables
  def __init__(self, inputs):
    self.inputs = inputs
    self.intermediate_output = []
    self.final_output = []
    self.class_classified = 0

  def calculating_output(self):
```

```
    #declaring variables res1 and res2 for storing result
    #declaring variables val1 and val2 for storing output before threshold is applied
    res1 = 0
    res2 = 0
    val1 = 0
    val2 = 0

    #adding the product input given to the neurons with the weight of the neurons
    #along with product of bias input and weight
    res1 = self.inputs[0]*(-2) + self.inputs[1]*(1) - (1 * (1/2))
    res2 = self.inputs[0]*(1) + self.inputs[1]*(-1) - (1 * (1/2))

    #thresholding the intermediate outputs to obtain binary final outputs
    #val1 and val2 are the final outputs after applying threshold
    if(res1 >= 0):
      val1 = 1

    else:
      val1 = -1

    #if result is negative the output is -1
    #if result is position the output is +1

    if(res2 >= 0):
      val2 = 1

    else:
      val2 = -1

    #finding the value of val1 + val2 + 1 for classifying the output in either Class 1
or Class 2
    #the weight and input of the bias is -1
    if val1 + val2 + (-1)*(-1) == -1:
      self.class_classified = 2

    elif val1 + val2 + 1 == 1:
      self.class_classified = 1

    #to store the output of both the neurons before the threshold was applied
    self.intermediate_output = [val1, val2]
```

```
    #inserting the value of val1 and val2 (outputs before threshold was applied) to the
array final_output[] for printing
    self.final_output = val1 + val2 + 1



print("Input", "      Output")

print()

#loop processes each input pattern and displays the intermediate outputs, final output,
and classified class
for i in inputs:

  #making an instance of the class Perceptron
  model = Perceptron(i)

  #function calculating_output() will be called for computing values
  model.calculating_output()
  print(i, "       ", model.final_output * (-1), "          ")
  print()
```

Output:

```
PS C:\Users\uditi> & C:/Users/u
_EXP2.py"
Input       Output

[1, 1]         1

[0, 0]         1

[1, 0]         -1

[0, 1]         -1
```

Date: _____ **Signature of faculty in-charge**