



<b>Name:</b> Heramb Ramakant Pawar	<b>Class/Roll No.:</b> D16AD/67	<b>Grade:</b>
------------------------------------	---------------------------------	---------------

**Title of Experiment:** Multi-layer Perceptron algorithm to Simulate XOR gate

**Objective of Experiment:**

The objective of this project is to build a Multi-Layer Perceptron neural network that can accurately simulate the XOR gate's behavior. The network should take two binary inputs (0 or 1) and produce corresponding binary outputs (0 or 1) that align with the XOR truth table.

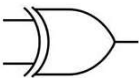
**Outcome of Experiment:** We successfully implemented XOR Gate Using Multi-Layer Perceptron

**Problem Statement:** Design and implement a Multi-Layer Perceptron neural network to simulate the XOR gate operation.

**Description / Theory:**

**XOR Gate:**

XOR gate (exclusive OR gate) is a basic electronic circuit that takes in two binary inputs, which can be either 0 or 1, and produces a single binary output. The output of the XOR gate is 1 when the number of 1s in the inputs is odd. If the number of 1s in the inputs is even, the output will be 0.

XOR	A	B	Output
	0	0	0
	1	0	1
	0	1	1
	1	1	0



## **Multi-Layer Perceptron:**

A Multi-Layer Perceptron (MLP) is a type of artificial neural network that belongs to the family of feed-forward neural networks. It is one of the foundational architectures in deep learning and is widely used for various tasks, such as classification, regression, and pattern recognition.

The term "Perceptron" originated from the work of Frank Rosenblatt in the late 1950s. The original Perceptron was a single-layer neural network capable of binary classification. However, the concept of a single layer was limited in its ability to learn complex patterns and non-linear relationships in data. The Multi-Layer Perceptron was introduced as an extension to overcome these limitations.

The main characteristic of a Multi-Layer Perceptron is its ability to stack multiple layers of neurons, also known as nodes or units. The network typically consists of three types of layers:

**Input Layer:** This is the first layer of the network, where the data is fed into the neural network. Each node in the input layer represents a feature or attribute of the input data.

**Hidden Layers:** These are intermediate layers between the input and output layers. Each node in the hidden layers takes inputs from the previous layer, performs some transformation using weights and biases, and then passes the result to the next layer. The presence of hidden layers allows Multi-Layer Perceptron to learn complex representations of data.

**Output Layer:** This is the final layer of the network that produces the desired output.

The connections between nodes in different layers are defined by weights, and each node also has an associated bias. During training, the Multi-Layer Perceptron adjusts these weights and biases using optimization algorithms like gradient descent in order to minimize the error or loss between the predicted output and the actual target.



**Program :**

```
import numpy as np
```

```
def unit_step(v):  
    return 1 if v >= 0 else 0
```

```
def perceptron_model(x, w, b):  
    v = np.dot(w, x) + b  
    y = unit_step(v)  
    return y
```

```
def NOT_logic_function(x):  
    w_NOT = -1  
    b_NOT = 0.5  
    return perceptron_model(x, w_NOT, b_NOT)
```

```
def AND_logic_function(x):  
    w_AND = np.array([1, 1])  
    b_AND = -1.5  
    return perceptron_model(x, w_AND, b_AND)
```

```
def OR_logic_function(x):  
    w_OR = np.array([1, 1])  
    b_OR = -0.5  
    return perceptron_model(x, w_OR, b_OR)
```

```
def XOR_logic_function(x):  
    y1 = AND_logic_function(x)  
    y2 = OR_logic_function(x)  
    y3 = NOT_logic_function(y1)  
    final_x = np.array([y2, y3])  
    return AND_logic_function(final_x)
```



```
test_cases = [  
    (0, 0),  
    (0, 1),  
    (1, 0),  
    (1, 1),  
]  
  
print("\n\n Results:")  
for test in test_cases:  
    x1, x2 = test  
    output = XOR_logic_function(np.array([x1, x2]))  
    print(f"\tXOR({x1}, {x2}) = {output}")
```

### Output Screenshots:

```
Results:  
XOR(0, 0) = 0  
XOR(0, 1) = 1  
XOR(1, 0) = 1  
XOR(1, 1) = 0
```



## **Results and Discussions:**

After training the Multi-Layer Perceptron on the XOR gate dataset, the model is evaluated based on the accuracy of its predictions. The model's performance is assessed by comparing the predicted output with the ground truth values of the XOR gate.

The Multi-Layer Perceptron algorithm proves to be capable of simulating the XOR gate successfully. Due to its ability to model non-linear relationships between inputs and outputs, the hidden layer in the Multi-Layer Perceptron allows it to learn the XOR function effectively.

The number of neurons in the hidden layer is a crucial hyper parameter that influences the performance of the Multi-Layer Perceptron. Too few neurons might lead to under fitting, where the model lacks the capacity to learn the XOR gate's complexity. On the other hand, too many neurons might lead to over fitting, causing the model to memorize the training data without generalizing well to new, unseen inputs.

With appropriate hyper parameter tuning, the Multi-Layer Perceptron can achieve high accuracy in simulating the XOR gate. It demonstrates the capability of artificial neural networks to handle non-linearly separable problems, making them powerful tools for various real-world applications.