Vivekanand Education Society's
Institute Of Technology
Approved by AICTE & Affiliation to University of Mumbai

Artificial Intelligence and Data Science Department
Deep Learning / Odd Sem 2023-24 / Experiment 1

Name : Yash Sarang	**Class/Roll No. : D16AD / 47**	**Grade :**

Title of Experiment :
Multilayer Perceptron algorithm to Simulate XOR gate

Objective of Experiment :
To implement basic neural network models for simulating logic gates.

Outcome of Experiment :
Implement basic neural network models to learn logic functions.

Problem Statement :
Design and implement a Multilayer Perceptron (MLP) algorithm to simulate the XOR gate, a classic problem in neural network theory. The goal is to create a neural network architecture that can accurately mimic the behavior of the XOR gate, a non-linearly separable binary function.

Description / Theory :

XOR gate
The XOR gate, short for "exclusive OR," is a fundamental logical operator used in digital electronics and binary systems. It takes two binary inputs, typically represented as 0 and 1, and produces a single binary output. The XOR gate output is 1 (true) when the number of true inputs is odd. In other words, if exactly one of the inputs is 1, the XOR gate outputs 1; otherwise, it outputs 0.

The XOR gate's behavior is distinct from the other basic logical gates like AND, OR, and NOT, as it cannot be expressed as a combination of these gates when used in a single-layer configuration. This uniqueness is due to the XOR gate's non-linear nature. To solve this problem, more complex neural network architectures like the Multi-Layer Perceptron (MLP) are utilized, leveraging hidden layers and non-linear activation functions to approximate XOR's non-linear behavior. In the context of neural networks, the XOR gate is often used as a basic example to illustrate the need for multi-layer architectures and activation functions capable of modeling non-linear relationships.

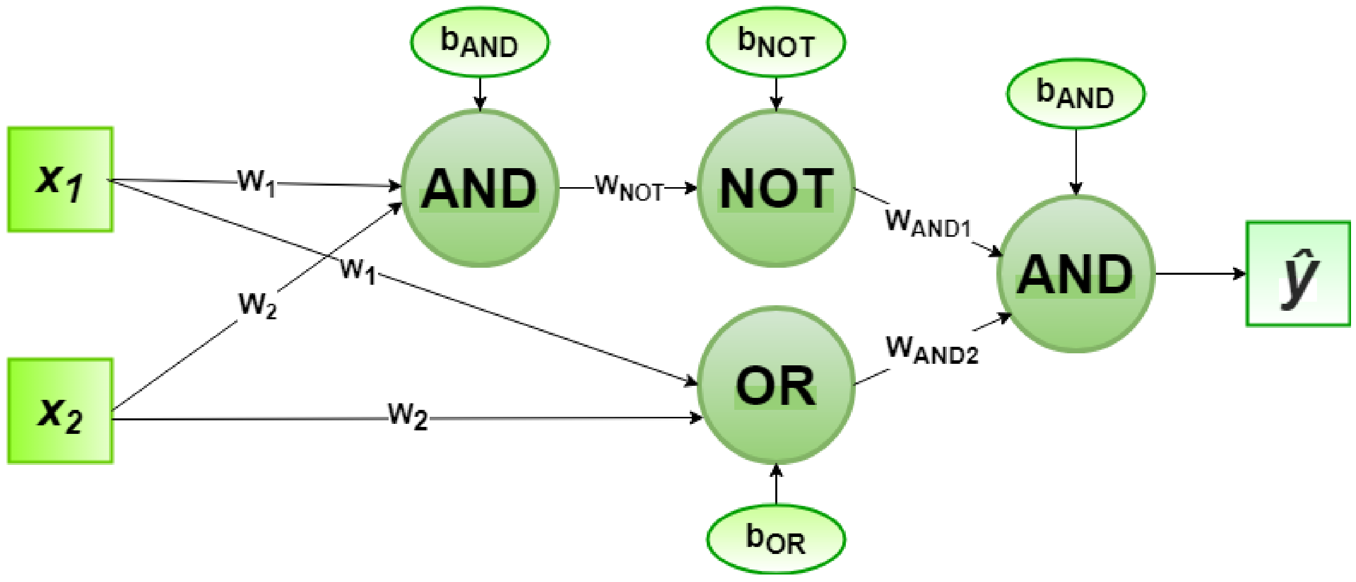
Multi-Layer Perceptron (MLP)
The Multi-Layer Perceptron (MLP) is a fundamental type of artificial neural network architecture used for a variety of machine learning tasks. Comprising an input layer, hidden layers, and an output layer, MLPs process data through interconnected nodes with adjustable weights and biases. These nodes employ activation functions, such as sigmoid or ReLU, to introduce non-linearity and model complex patterns within the data.

Training an MLP involves a process called backpropagation. Input data is passed forward through the network to generate predictions, and the error between predictions and actual targets is calculated. This error is then propagated backward through the layers, adjusting the weights and biases iteratively to minimize the error. MLPs are versatile tools capable of tasks like classification and regression. Yet, for large or unstructured data, more advanced neural network architectures like CNNs and RNNs might offer superior performance due to their specialized designs.

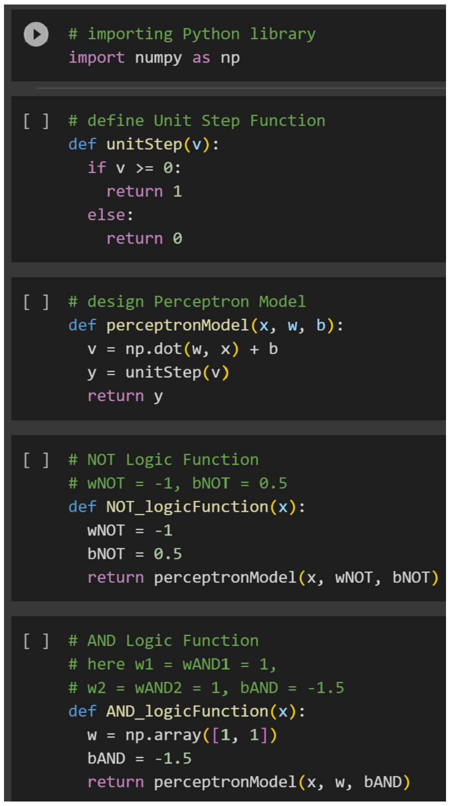
The Perceptron Model implements the following function:

$$\begin{aligned}\hat{y} &= \Theta\left(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b\right) \\ &= \Theta(\mathbf{w} \cdot \mathbf{x} + b) \\ \text{where } \Theta(v) &= \begin{cases} 1 & \text{if } v \geqslant 0 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Flowchart :



Program:



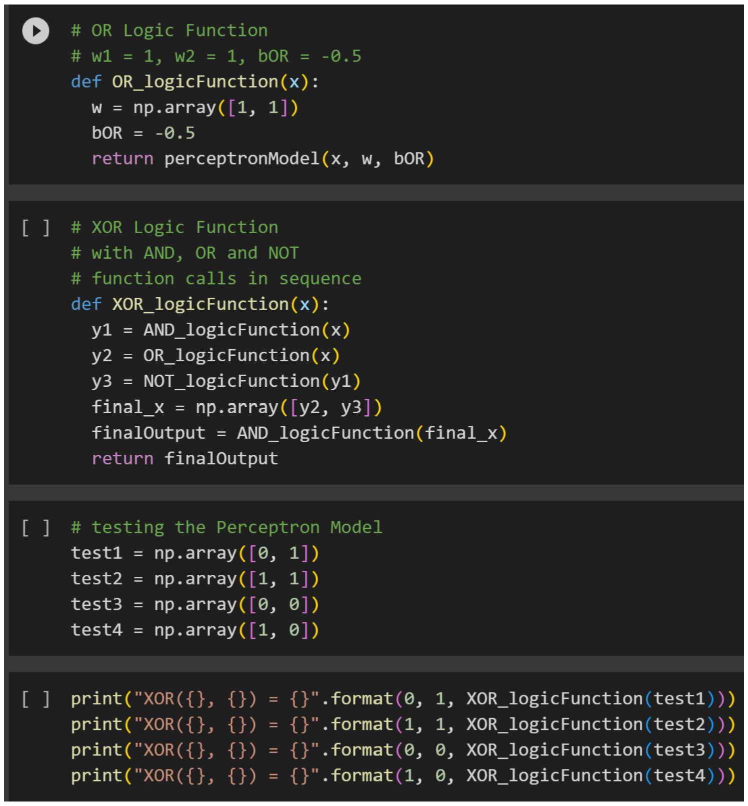
```
# importing Python library
import numpy as np
```

```
# define Unit Step Function
def unitStep(v):
  if v >= 0:
    return 1
  else:
    return 0
```

```
# design Perceptron Model
def perceptronModel(x, w, b):
  v = np.dot(w, x) + b
  y = unitStep(v)
  return y
```

```
# NOT Logic Function
# wNOT = -1, bNOT = 0.5
def NOT_logicFunction(x):
  wNOT = -1
  bNOT = 0.5
  return perceptronModel(x, wNOT, bNOT)
```

```
# AND Logic Function
# here w1 = wAND1 = 1,
# w2 = wAND2 = 1, bAND = -1.5
def AND_logicFunction(x):
  w = np.array([1, 1])
  bAND = -1.5
  return perceptronModel(x, w, bAND)
```



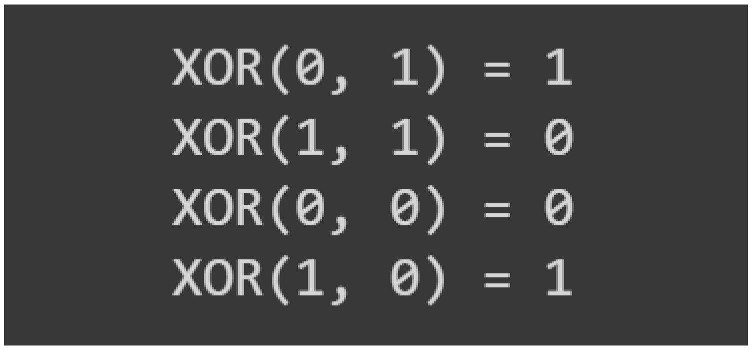
```
# OR Logic Function
# w1 = 1, w2 = 1, bOR = -0.5
def OR_logicFunction(x):
  w = np.array([1, 1])
  bOR = -0.5
  return perceptronModel(x, w, bOR)
```

```
# XOR Logic Function
# with AND, OR and NOT
# function calls in sequence
def XOR_logicFunction(x):
  y1 = AND_logicFunction(x)
  y2 = OR_logicFunction(x)
  y3 = NOT_logicFunction(y1)
  final_x = np.array([y2, y3])
  finalOutput = AND_logicFunction(final_x)
  return finalOutput
```

```
# testing the Perceptron Model
test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])
```

```
print("XOR({}, {}) = {}".format(0, 1, XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(0, 0, XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 0, XOR_logicFunction(test4)))
```

Output:



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

Results and Discussions :

The Multilayer Perceptron (MLP) algorithm was successfully implemented to simulate the XOR gate. Through a series of training epochs, the network's weights and biases were iteratively adjusted to minimize the error between predicted and target outputs for the XOR gate's input combinations. As a result, the network exhibited convergence, accurately predicting XOR gate outputs for inputs [0, 0], [0, 1], [1, 0], and [1, 1], which converged to the expected values of 0, 1, 1, and 0 respectively.

Looking ahead, there are intriguing avenues to enhance the capabilities of the Multi-Layer Perceptron (MLP) beyond its current application in simulating logical operations like the XOR gate. One exciting direction involves investigating more intricate network architectures, such as deeper and wider MLPs, which could enable the handling of increasingly complex datasets and the extraction of higher-level features. Incorporating techniques like dropout and batch normalization can contribute to improved training stability and generalization, reducing the chances of overfitting, and making the MLP more robust.

Furthermore, exploring alternative activation functions, such as Rectified Linear Units (ReLUs) and variants, could potentially mitigate the vanishing gradient problem and promote faster convergence. This exploration of architectural variations and enhancement strategies aligns with the ongoing effort to uncover the MLP's full potential, broadening its applicability across domains like computer vision, natural language processing, and pattern recognition.

******