DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

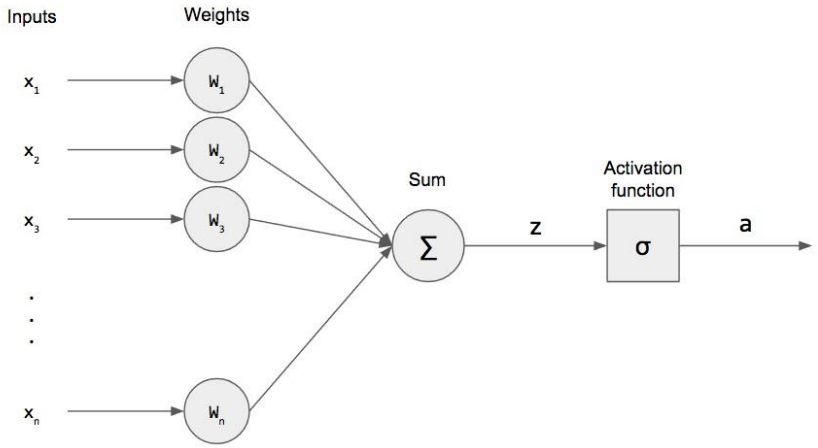
LAB EXPERIMENT NO.1

AIM :

Implement Boolean gates using perceptron – Neural representation of Logic Gates.

THEORY:

Perceptron is a Supervised Learning Algorithm for binary classifiers.



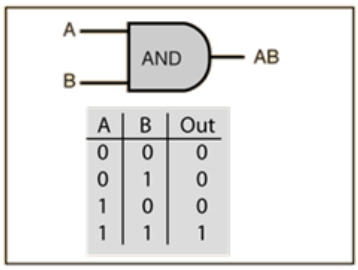
For a particular choice of the weight vector w and bias parameter b, the model predicts output ŷ for the corresponding input vector x.

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

The input values, i.e., x1, x2, and bias is multiplied with their respective weight matrix that is W1, W2, and W0. The corresponding value is then fed to the summation neuron where the summed value is calculated. This is fed to a neuron which has a non-linear function(sigmoid in our case) for scaling the output to a desirable range. The scaled output of sigmoid is 0 if the output is less than 0.5 and 1 if the output is greater than 0.5. The main aim is to find the value of weights or the weight vector which will enable the system to act as a particular gate.

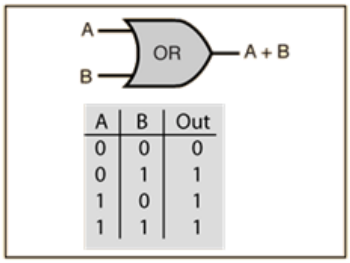
Boolean gates – Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic.

1) AND



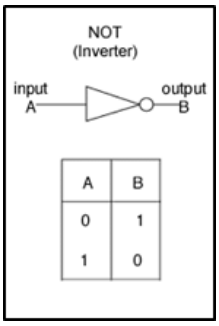
A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

2) OR



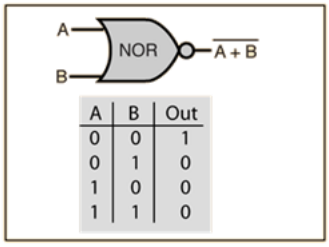
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

3) NOT



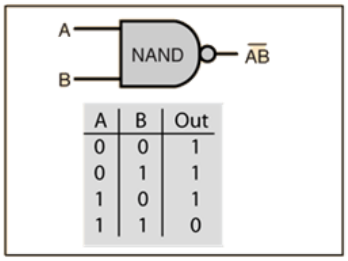
A	B
0	1
1	0

4) NOR



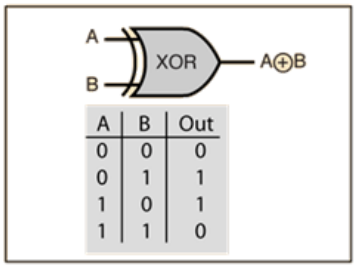
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

5) NAND



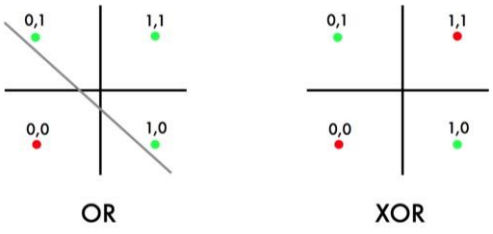
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

6) XOR



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

For the XOR gate, the output can't be linearly separated. Uni layered perceptrons can only work with linearly separable data. But in the following diagram drawn in accordance with the truth table of the XOR logical operator, we can see that the data is NOT linearly separable.



To solve this problem, we add an extra layer to our vanilla perceptron, i.e., we create a Multi Layered Perceptron (or MLP). We call this extra layer as the Hidden layer. To build a perceptron, we first need to understand that the XOR gate can be written as a combination of AND gates, NOT gates and OR gates in the following way:

$$XOR(\boldsymbol{x_1}, \boldsymbol{x_2}) = AND(NOT(AND(\boldsymbol{x_1}, \boldsymbol{x_2})), OR(\boldsymbol{x_1}, \boldsymbol{x_2}))$$

Hidden layers are those layers with nodes other than the input and output nodes. An MLP is generally restricted to having a single hidden layer. The hidden layer allows for non-linearity. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with their own weights and biases, and an output is computed, generally with an activation function.

1. **Implement the 6 Boolean gates above using perceptrons. Inputs = x1, x2 and bias, weights should be fed into the perceptron with single Output = y**
2. **Display final weights and bias of each perceptron.**
3. **What are the limitations of the perceptron network?**
4. **Implement AND, OR, NAND, NOR, NOT and XOR gate implementation.**

Name : AYUSHI SINGH

SAP ID : 60009220202

```
import pandas as pd
import numpy as np
```

```
class Neuron():
    def __init__(self, x):
        self.x = x
    def perceptron(self, x, w, b):
        try:             y_ =
np.dot(x, w) + b
        except:
            y_ = np.dot(w, x) + b
        print("Weights = ", w , "Bias = ", b)
        return y_
    def step_func(self, y_ ):
        return y_ >= 0
def and_func(self, x):
        w = np.array([1,1])
        b = -2
        y_ = self.perceptron(x, w, b)
        return y_ >= 0
def or_func(self, x):
        w = np.array([1,1])
        b = -0.5
        y_ = self.perceptron(x, w, b)
return self.step_func(y_)
    def not_func(self, x):
        w = -1
        b = 0.5
        y_ = self.perceptron(x, w, b)
return self.step_func(y_)
    def nand_func(self, x):
        o_and = self.and_func(x)
        y_ = self.not_func(o_and)
        return y_
    def nor_func(self, x):
        o_or = self.or_func(x)
        y_ = self.not_func(o_or)
        return y_
    def xor_func(self, x):
        nand = self.nand_func(x)
        or_ = self.or_func(x)
        y_ = self.and_func(np.array([or_, nand]))
        return y_
```

```
ok = Neuron([[1,0],[0,1],[1,1],[0,0]])
print(ok.or_func([[1,0],[0,1],[1,1],[0,0]]))
```

```
Weights =  [1 1] Bias =  -0.5
    [ True  True  True False]
```

```
print(ok.and_func([[1,0],[0,1],[1,1],[0,0]]))
```

```
Weights =  [1 1] Bias =  -2
    [False False  True False]
```

```
print(ok.nand_func([[1,0],[0,1],[1,1],[0,0]]))
```

```
Weights =  [1 1] Bias =  -2
    Weights =  -1 Bias =  0.5
    [ True  True False  True]
```

```
print(ok.not_func([[1],[0]]))
```

```
Weights =  -1 Bias =  0.5
    [[False]
     [ True]]
```

```
print(ok.or_func([[1,0],[0,1],[1,1],[0,0]]))
```

```
Weights =  [1 1] Bias =  -0.5
    [ True  True  True False]
```

```
print(ok.nor_func([[1,0],[0,1],[1,1],[0,0]]))
```

```
Weights =  [1 1] Bias =  -0.5
    Weights =  -1 Bias =  0.5
    [False False False  True]
```

```
print(ok.xor_func(np.array([[1,0],[0,1],[1,1],[0,0]])))
```

```
Weights =  [1 1] Bias =  -2
 Weights =  -1 Bias =  0.5
 Weights =  [1 1] Bias =  -0.5
 Weights =  [1 1] Bias =  -2
 [ True  True False False]
```

❓ Limitations of perceptrons :

Linearly Separable Problems Only: Single-layer perceptrons can only solve problems that are linearly separable. This means that they can only classify datasets where a straight line (in 2D) or a

hyperplane (in higher dimensions) can separate the classes. This is a signi cant limitation for more complex problems like XOR, which are not linearly separable.

Limited to Simple Tasks: Perceptrons are limited in their ability to model complex relationships in data due to their simple structure. Tasks requiring nonlinear decision boundaries, like those needed in XOR and XNOR gates, cannot be effectively modeled with a single-layer perceptron.

Binary Classi cation: Perceptrons are inherently designed for binary classi cation problems. While they can be adapted for multiclass classi cation (using methods like one-vs-all), their natural application is limited to distinguishing between two classes.

No Learning of Non-Linear Patterns: A perceptron cannot capture non-linear patterns or interactions between features. This requires the introduction of hidden layers, leading to more complex neural networks like multi-layer perceptrons (MLPs).

Conclusion

In this experiment, we successfully implemented simple gates like AND, OR, NAND, NOT, XOR and NOR. The perceptron network, despite its simplicity, plays a fundamental role in the development of neural network theory. It is an effective model for solving simple, linearly separable problems and serves as a building block for more complex networks. While its limitations, particularly the inability to solve non-linear problems like XOR, restrict its applicability, the perceptron's strengths lie in its simplicity, interpretability, and e ciency.