SAP ID:

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

NAME: Divyesh Khunt **SAPID:60009210116** **Batch:D12**

COURSE CODE: DJ19DSC501 **DATE:**

COURSE NAME: Machine Learning - II **CLASS: AY 2022-23**

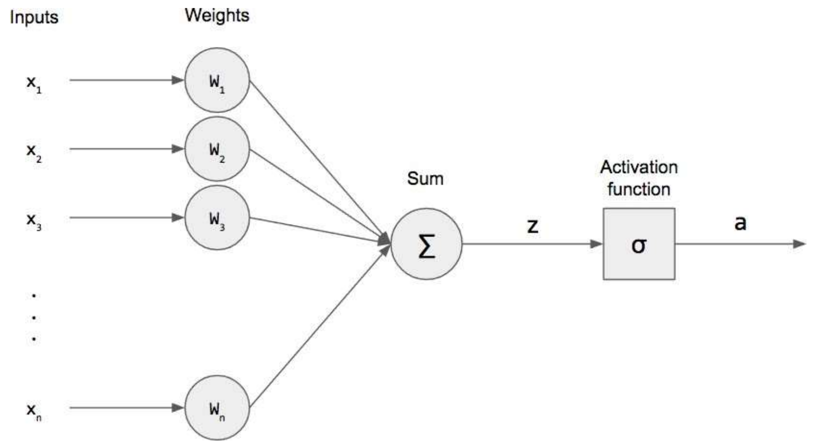
LAB EXPERIMENT NO.1

AIM :

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

THEORY:

Perceptron is a Supervised Learning Algorithm for binary classifiers.



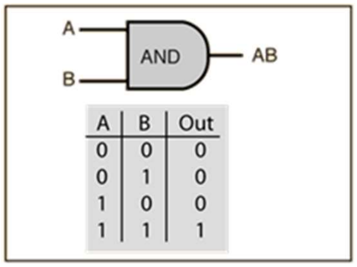
For a particular choice of the weight vector w and bias parameter b, the model predicts output ŷ for the corresponding input vector x.

$$\begin{aligned}\hat{y} &= \Theta\left(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b\right)\\ &= \Theta(\mathbf{w}\cdot\mathbf{x} + b)\\ \text{where } \Theta(v) &= \begin{cases} 1 & \text{if } v \geqslant 0 \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

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.

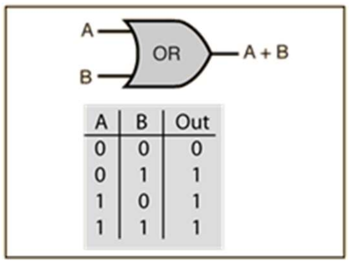
Boolean gates – Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic.

1) AND



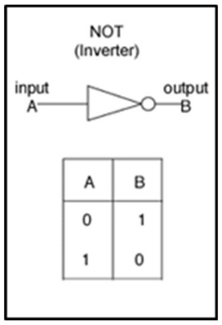
A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

2) OR



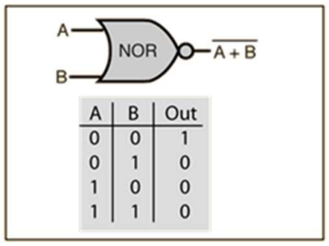
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

3) NOT



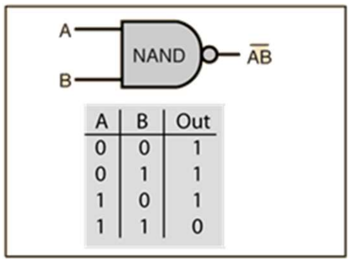
A	B
0	1
1	0

4) NOR



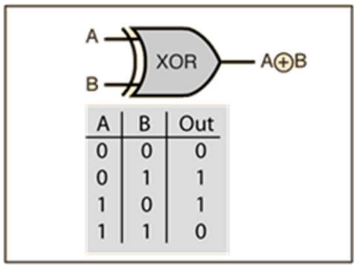
A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

5) NAND



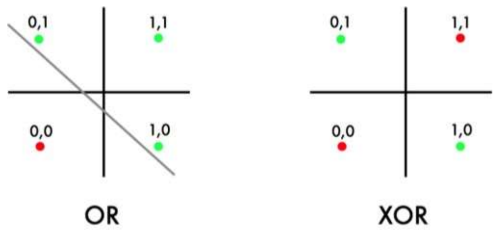
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

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

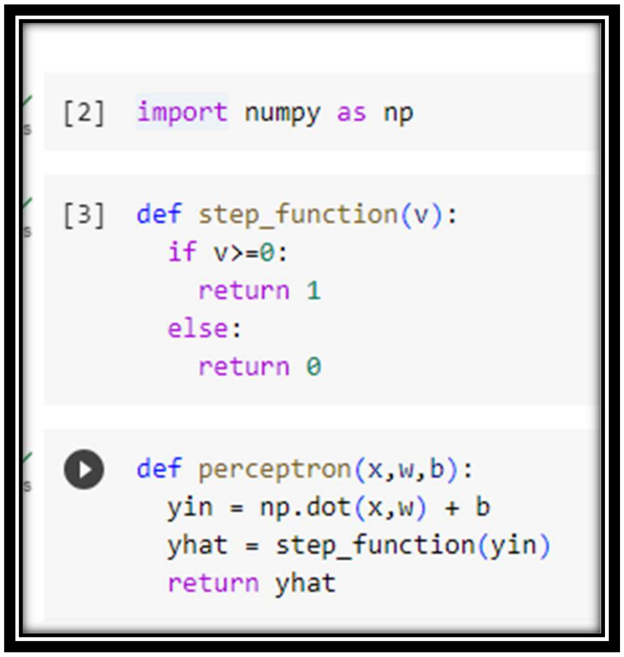
Hidden layers are those layers with nodes other than the input and output nodes. An MLP is generally restricted to having a single hidden layer. The hidden layer allows for non-linearity. A node in the hidden layer isn't too different to an output node: nodes in the previous layers connect to it with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

CONCLUSION:

Base all conclusions on your actual results; describe the meaning of the experiment and the implications of your results.

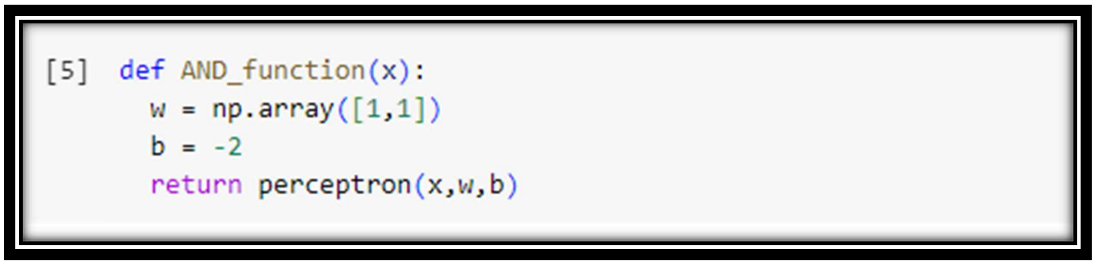
1.,2.



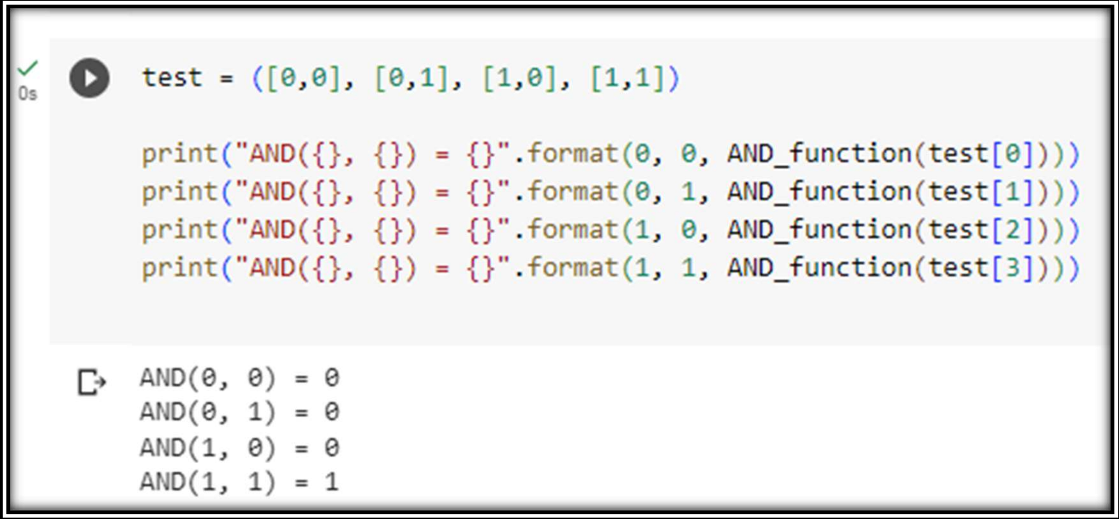
```
[2]  import numpy as np

[3]  def step_function(v):
       if v>=0:
         return 1
       else:
         return 0

     def perceptron(x,w,b):
       yin = np.dot(x,w) + b
       yhat = step_function(yin)
       return yhat
```



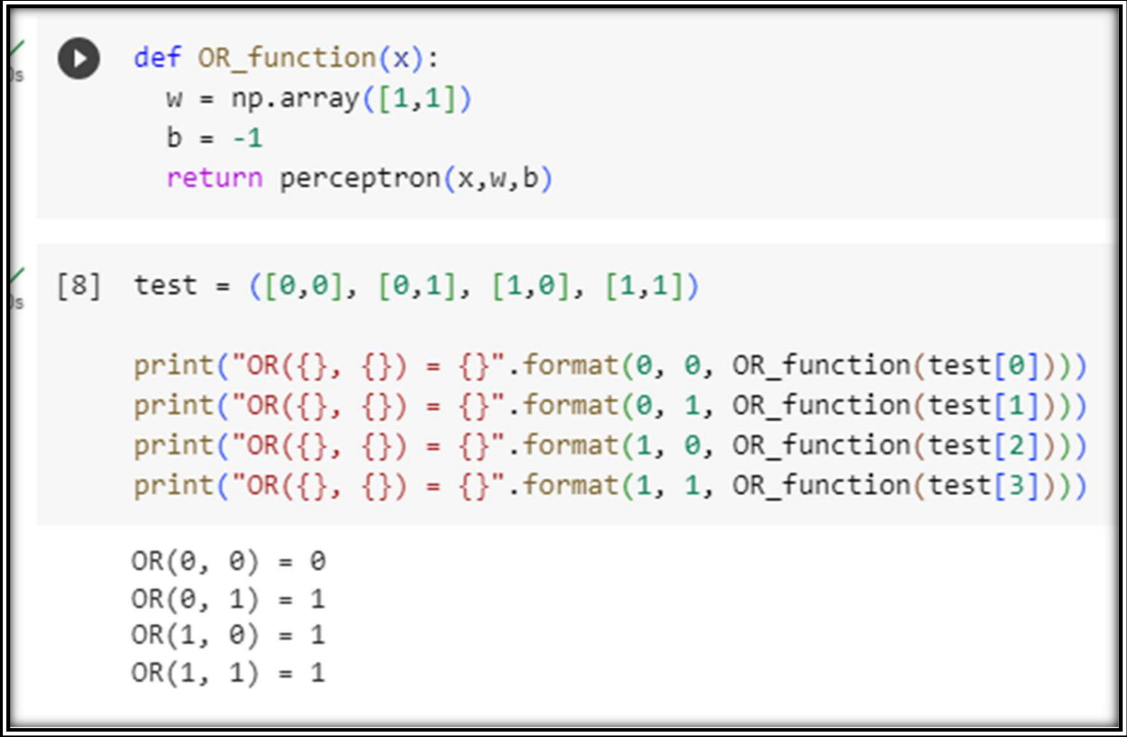
```
[5]  def AND_function(x):
       w = np.array([1,1])
       b = -2
       return perceptron(x,w,b)
```



```
test = ([0,0], [0,1], [1,0], [1,1])

print("AND({}, {}) = {}".format(0, 0, AND_function(test[0])))
print("AND({}, {}) = {}".format(0, 1, AND_function(test[1])))
print("AND({}, {}) = {}".format(1, 0, AND_function(test[2])))
print("AND({}, {}) = {}".format(1, 1, AND_function(test[3])))
```

```
AND(0, 0) = 0
AND(0, 1) = 0
AND(1, 0) = 0
AND(1, 1) = 1
```

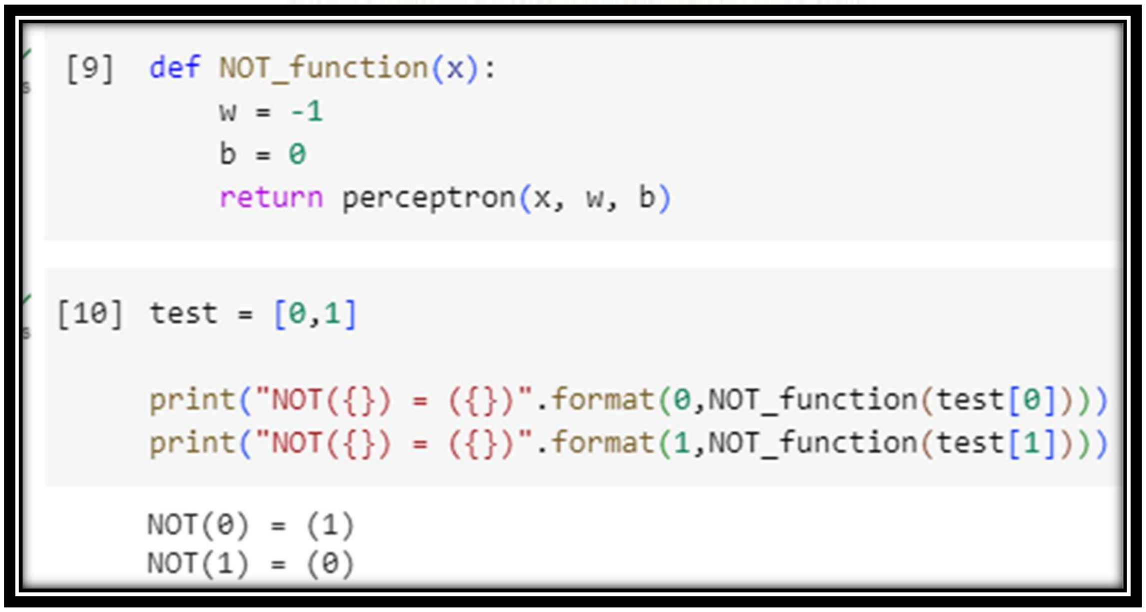


```
def OR_function(x):
  w = np.array([1,1])
  b = -1
  return perceptron(x,w,b)
```

```
[8] test = ([0,0], [0,1], [1,0], [1,1])

print("OR({}, {}) = {}".format(0, 0, OR_function(test[0])))
print("OR({}, {}) = {}".format(0, 1, OR_function(test[1])))
print("OR({}, {}) = {}".format(1, 0, OR_function(test[2])))
print("OR({}, {}) = {}".format(1, 1, OR_function(test[3])))
```

```
OR(0, 0) = 0
OR(0, 1) = 1
OR(1, 0) = 1
OR(1, 1) = 1
```

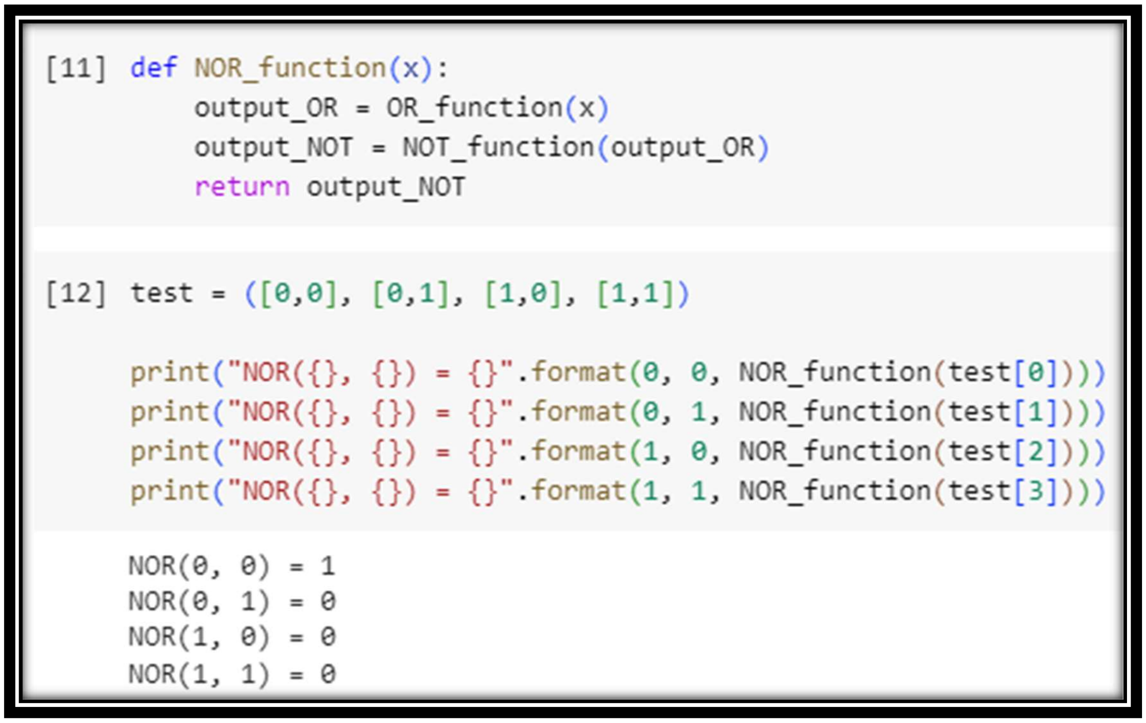


```
[9]  def NOT_function(x):
         w = -1
         b = 0
         return perceptron(x, w, b)

[10] test = [0,1]

     print("NOT({}) = ({})".format(0,NOT_function(test[0])))
     print("NOT({}) = ({})".format(1,NOT_function(test[1])))

     NOT(0) = (1)
     NOT(1) = (0)
```

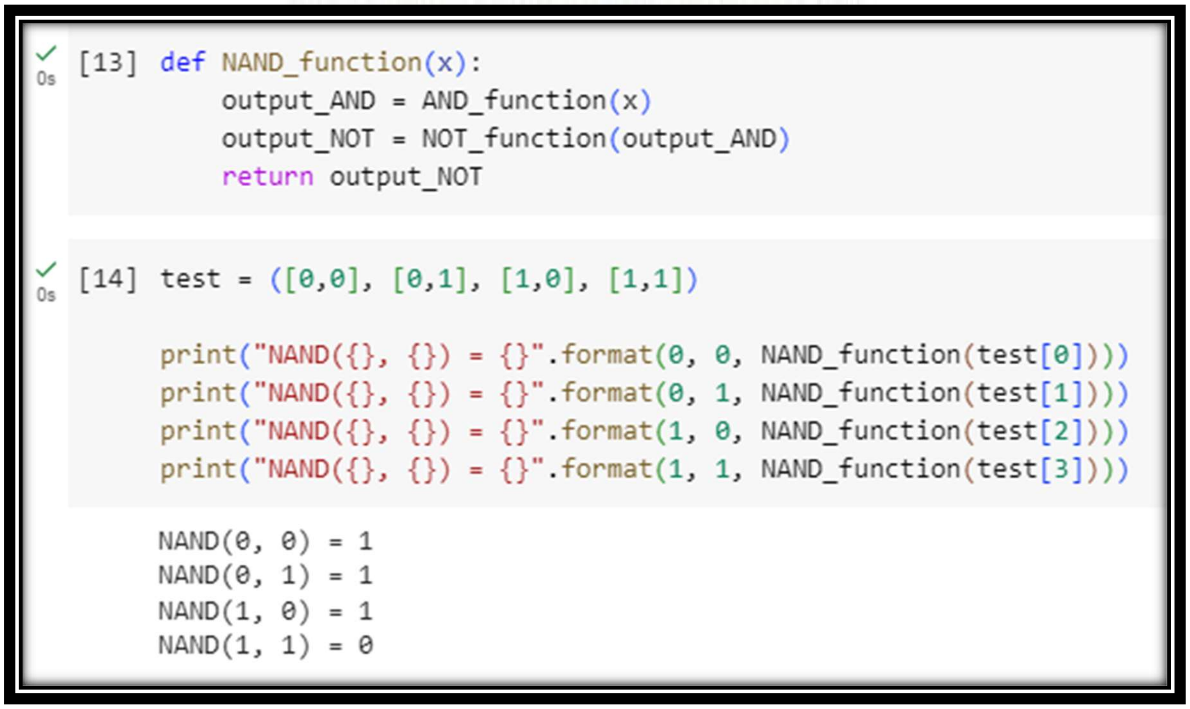


```
[11] def NOR_function(x):
         output_OR = OR_function(x)
         output_NOT = NOT_function(output_OR)
         return output_NOT

[12] test = ([0,0], [0,1], [1,0], [1,1])

     print("NOR({}, {}) = {}".format(0, 0, NOR_function(test[0])))
     print("NOR({}, {}) = {}".format(0, 1, NOR_function(test[1])))
     print("NOR({}, {}) = {}".format(1, 0, NOR_function(test[2])))
     print("NOR({}, {}) = {}".format(1, 1, NOR_function(test[3])))

     NOR(0, 0) = 1
     NOR(0, 1) = 0
     NOR(1, 0) = 0
     NOR(1, 1) = 0
```

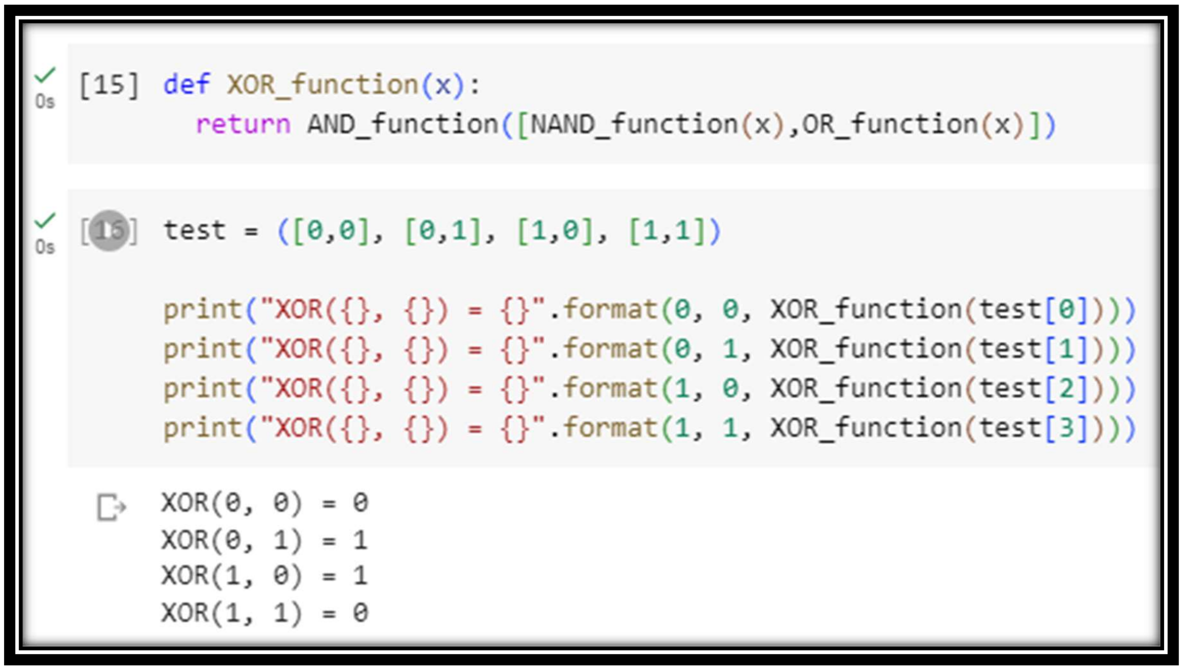


```
[13] def NAND_function(x):
         output_AND = AND_function(x)
         output_NOT = NOT_function(output_AND)
         return output_NOT
```

```
[14] test = ([0,0], [0,1], [1,0], [1,1])

     print("NAND({}, {}) = {}".format(0, 0, NAND_function(test[0])))
     print("NAND({}, {}) = {}".format(0, 1, NAND_function(test[1])))
     print("NAND({}, {}) = {}".format(1, 0, NAND_function(test[2])))
     print("NAND({}, {}) = {}".format(1, 1, NAND_function(test[3])))
```

```
NAND(0, 0) = 1
NAND(0, 1) = 1
NAND(1, 0) = 1
NAND(1, 1) = 0
```



```
[15] def XOR_function(x):
       return AND_function([NAND_function(x),OR_function(x)])
```

```
[16] test = ([0,0], [0,1], [1,0], [1,1])

     print("XOR({}, {}) = {}".format(0, 0, XOR_function(test[0])))
     print("XOR({}, {}) = {}".format(0, 1, XOR_function(test[1])))
     print("XOR({}, {}) = {}".format(1, 0, XOR_function(test[2])))
     print("XOR({}, {}) = {}".format(1, 1, XOR_function(test[3])))
```

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

3**. The perceptron network, while a foundational concept in neural network history, has several limitations:**

1. One limitation of perceptrons is that they can only learn functions that have linear decision boundaries. This means they are unable to solve problems that require non-linear decision making, such as the XOR problem. For more complex tasks, a single perceptron alone is insufficient.
2. Perceptrons, unlike more advanced models, only have a single layer and lack the capability to capture complex relationships in data. This limits their effectiveness in solving intricate problems or representing hierarchical structures.
3. RephraseOne limitation of perceptrons is that they produce binary outputs, either 0 or 1. This can restrict their ability to express more nuanced information. In contrast, modern neural networks utilize activation functions that allow for continuous outputs, enabling a richer encoding of data.
4. Perceptrons are highly sensitive to the scale of input features, which means that data preprocessing is crucial. However, meticulously preprocessing large or high-dimensional datasets can be impractical.
5. RephraseLimited Representation: AI systems have difficulty capturing features and representations at various levels of abstraction. This is crucial for tasks like image recognition and natural language processing.
6. RephraseTraining Difficulties: Perceptrons face challenges when dealing with complex problems or those that necessitate precise adjustments. Their reliance on a basic update rule, such as the perceptron learning rule, can hinder their ability to converge on optimal solutions.

To overcome these limitations, researchers developed multilayer perceptrons, also known as feedforward neural networks. By introducing hidden layers and non-linear activation functions, they were able to address the shortcomings of perceptrons and effectively handle various tasks. This development played a crucial role in paving the way for the deep learning revolution.