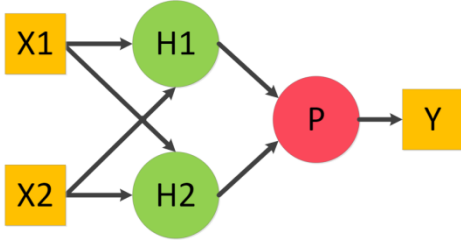
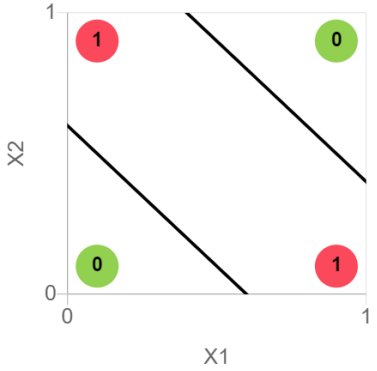


Experiment No. 2																				
BE (AI&DS)		ROLL NO : 9742																		
Date of Implementation: 26/07/2024																				
Aim: Implement Multilayer Perceptron algorithm to simulate XOR gate.																				
Programming Language Used : Python																				
Upon completion of this experiment, students will be able to LO1 : Implement basic neural network model for a given a problem.																				
<table border="1"> <tr> <td>Indicator</td> <td></td> <td></td> </tr> <tr> <td>Timeline</td> <td></td> <td></td> </tr> <tr> <td>Maintains submission deadline (1)</td> <td>On time (1)</td> <td>Otherwise (0)</td> </tr> <tr> <td>Completion and Organization (2)</td> <td>Completed in LAB (2)</td> <td>Otherwise(1)</td> </tr> <tr> <td>Analysis of output and conclusion(2)</td> <td>Properly done (2)</td> <td>Otherwise (0)</td> </tr> <tr> <td>Viva (10)</td> <td></td> <td></td> </tr> </table>			Indicator			Timeline			Maintains submission deadline (1)	On time (1)	Otherwise (0)	Completion and Organization (2)	Completed in LAB (2)	Otherwise(1)	Analysis of output and conclusion(2)	Properly done (2)	Otherwise (0)	Viva (10)		
Indicator																				
Timeline																				
Maintains submission deadline (1)	On time (1)	Otherwise (0)																		
Completion and Organization (2)	Completed in LAB (2)	Otherwise(1)																		
Analysis of output and conclusion(2)	Properly done (2)	Otherwise (0)																		
Viva (10)																				
Assessment Marks : <table border="1"> <tr> <td>Timeline(1)</td> <td></td> </tr> <tr> <td>Completion and Organization (2)</td> <td></td> </tr> <tr> <td>Analysis of output and conclusion(2)</td> <td></td> </tr> <tr> <td>Viva (10)</td> <td></td> </tr> <tr> <td>Total (15)</td> <td></td> </tr> </table>			Timeline(1)		Completion and Organization (2)		Analysis of output and conclusion(2)		Viva (10)		Total (15)									
Timeline(1)																				
Completion and Organization (2)																				
Analysis of output and conclusion(2)																				
Viva (10)																				
Total (15)																				

EXPERIMENT	2
Aim	To Implement Multilayer Perceptron algorithm to simulate XOR gate.
Tools	PYTHON
Theory	<p>XOR function returns true if either but not both input values are true (1), otherwise returning false (0). Although seemingly straightforward when working with linearly separable data, traditional binary classifiers such as single-layer perceptron struggle to perform accurately on XOR-like problems due to non-linear decision boundaries. Designed by Frank Rosenblatt in 1958, the perceptron algorithm revolutionized early AI research. It mimics biological neurons found in human brains while capitalizing on mathematical principles to make accurate predictions or decisions based on input patterns. The simplest kind of neural network is a single-layer perceptron, consisting of a single node P, two inputs X1 and X2, an optional bias value, and a single output Y. The sum of the weighted input products and bias is calculated and fed into the node's specific activation function to produce the final output. The multi-layer perceptron has one or more additional hidden stacked layers, in this case one additional layer consisting of the H1 and H2 nodes. The individual nodes H1, H2 and P work as single-layer perceptrons, where each layer propagates its outputs to the inputs of the next layer. Typically nodes in a specific layer will use the same activation function.</p>  <p>A single-layer perceptron model cannot solve the XOR function since a single straight line cannot be drawn to separate and group the output patterns. However it is possible to draw two straight lines to separate and group the output patterns. A multi-layer perceptron containing an extra layer of hidden neurons is capable of solving problems in 3-dimensional hyperspace such as the XOR problem. The data is now linearly separable using a 2-dimensional hyperplane.</p>

	
Implementation	<p>Step 1: Define the input binary values (0 and 1) for all possible combinations. For XOR gate, the four inputs are (0, 0), (0, 1), (1, 0), and (1, 1).</p> <p>Step 2: Assign initial random weights and bias values – As a starting point in training our perceptron algorithm, assigning random weights between -1 and +1 is customary.</p> <p>Step 3: Train the Perceptron by adjusting weights accordingly – By calculating predicted output via weighted sum with an activation function applied – typically utilizing a threshold-based step function</p> <p>Step 4: Evaluate Training Results – After multiple iterations of training data sampling and weight adjustments based on predictions versus expected outputs, check model performance by comparing resultant predictions against the actual XOR logic table.</p>
Conclusion	<p>The single-layer perceptron cannot solve the XOR problem because it's not linearly separable. To solve XOR and other non-linearly separable problems, a multi-layer perceptron with hidden layers is necessary, as it can learn and represent complex decision boundaries. Thus, we have implemented Multilayer Perceptron algorithm to simulate XOR gate.</p>

Implementation:

```
In [17]: import numpy as np
```

```
In [18]: # Input and output data
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
expected_output = np.array([0, 1, 1, 0])
```

```
In [19]: # Initialize weights randomly between -1 and 1
hidden_layer1_weights = np.random.uniform(-1, 1, 2)
hidden_layer2_weights = np.random.uniform(-1, 1, 2)
output_layer_weights = np.random.uniform(-1, 1, 2)
```

```
In [20]: # Learning rate
eta = 1.5
```

```
In [21]: # Activation function: Step function
def activation_func(x):
    return 1 if x >= 1 else 0
```

```
In [22]: # Perceptron function
def perceptron(input_vals, weights):
    return activation_func(np.dot(input_vals, weights))
```

```
In [24]: # Store weights history
history_of_weights = []
iteration_count = 0
```

```
In [27]: while True:
    all_correct = True
    iteration_count += 1

    history_of_weights.append({
        'hidden1_weights': hidden_layer1_weights.copy(),
        'hidden2_weights': hidden_layer2_weights.copy(),
        'output_weights': output_layer_weights.copy()
    })

    for i in range(len(input_data)):
        x1, x2 = input_data[i]
        not_x1 = 1 - x1
        not_x2 = 1 - x2

        z1_input = np.array([x1 * not_x2, x1 * not_x1])
        z2_input = np.array([not_x1 * x2, x2 * not_x2])

        z1_output = perceptron(z1_input, hidden_layer1_weights)
        z2_output = perceptron(z2_input, hidden_layer2_weights)

        final_input = np.array([z1_output, z2_output])
        y = np.dot(final_input, output_layer_weights)
        prediction = activation_func(y)

        error = expected_output[i] - prediction

        if z1_output != expected_output[i]:
            hidden_layer1_weights += eta * error * z1_input

        if z2_output != expected_output[i]:
            hidden_layer2_weights += eta * error * z2_input

        if prediction != expected_output[i]:
            output_layer_weights += eta * error * final_input
```

```

        if prediction != expected_output[i]:
            all_correct = False

    if all_correct:
        break

    # Print weights for the first 5 iterations
    if iteration_count <= 5:
        print(f"\nWeights for iteration {iteration_count}:")
        print(f"  Weights for z1: {history_of_weights[-1]['hidden1_weights']}")
        print(f"  Weights for z2: {history_of_weights[-1]['hidden2_weights']}")
        print(f"  Weights for final output: {history_of_weights[-1]['output_weights']}")

```

```

In [28]: print("Final Weights for z1:")
print(f"w11: {hidden_layer1_weights[0]}, w12: {hidden_layer1_weights[1]}")

print("Final Weights for z2:")
print(f"w21: {hidden_layer2_weights[0]}, w22: {hidden_layer2_weights[1]}")

print("Weights between z1 and final output:")
print(f"v1: {output_layer_weights[0]}")

print("Weights between z2 and final output:")
print(f"v2: {output_layer_weights[1]}")

```

```

Final Weights for z1:
w11: 2.277384419908733, w12: 0.26732301108396084
Final Weights for z2:
w21: 1.0028513129626728, w22: -0.21301135376120683
Weights between z1 and final output:
v1: 1.107635509573699
Weights between z2 and final output:
v2: 1.779486729559178

```

```

In [29]: # Final predictions
final_predictions = []
for i in range(len(input_data)):
    x1, x2 = input_data[i]
    not_x1 = 1 - x1
    not_x2 = 1 - x2

    z1_input = np.array([x1 * not_x2, x1 * not_x1])
    z2_input = np.array([not_x1 * x2, x2 * not_x2])

    z1_output = perceptron(z1_input, hidden_layer1_weights)
    z2_output = perceptron(z2_input, hidden_layer2_weights)

    final_input = np.array([z1_output, z2_output])
    y = np.dot(final_input, output_layer_weights)
    final_predictions.append(activation_func(y))

print("Final Predictions:", final_predictions)
print("Expected Outputs:", expected_output.tolist())
print(f"Number of iterations: {iteration_count}")

```

```

Final Predictions: [0, 1, 1, 0]
Expected Outputs: [0, 1, 1, 0]
Number of iterations: 5

```

```

In [30]: # Print weights for the first and last iteration
print("\nWeights for the first iteration:")
print(f"Iteration 1:")
print(f"  Weights for z1: {history_of_weights[0]['hidden1_weights']}")
print(f"  Weights for z2: {history_of_weights[0]['hidden2_weights']}")
print(f"  Weights for final output: {history_of_weights[0]['output_weights']}")

```

```
print("\nWeights for the last iteration:")
print(f"Iteration {iteration_count}:")
print(f"  Weights for z1: {history_of_weights[-1]['hidden1_weights']}")
print(f"  Weights for z2: {history_of_weights[-1]['hidden2_weights']}")
print(f"  Weights for final output: {history_of_weights[-1]['output_weights']}")
```

Weights for the first iteration:

Iteration 1:

Weights for z1: [-0.72261558 0.26732301]

Weights for z2: [-0.49714869 -0.21301135]

Weights for final output: [-0.39236449 0.27948673]

Weights for the last iteration:

Iteration 5:

Weights for z1: [2.27738442 0.26732301]

Weights for z2: [1.00285131 -0.21301135]

Weights for final output: [1.10763551 1.77948673]