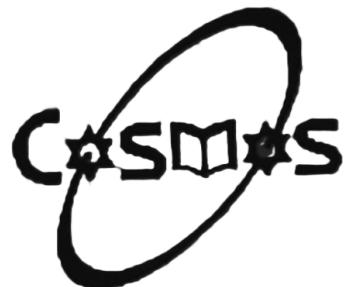


**Cosmos College of  
Management & Technology**  
(Affiliated to Pokhara University)

Sitapaila, Kathmandu



**Artifical Intelligence (CMP 346) Lab Report**

**LAB REPORT NO: 07**

**LAB REPORT ON:**  
PERCEPTRON

**SUBMITTED BY:-**

**Name:** Kushal Prasad Joshi

**Roll No:** 230345

**Faculty:** Science and Technology

**Group:** B

**SUBMITTED TO:-**

**Department:** ICT

**Lab Instructor:** Mr. Ranjan Raj Aryal

**Date of Experiment:** 23rd Janaury 2026

**Date of Submission:** 1st February 2026

## TITLE

PERCEPTRON

## OBJECTIVES

1. To understand the basic concepts of Perceptron.
2. To implement single layer perceptron.
3. To implement multi layer perceptron.

## REQUIREMENTS

- Python
- VS Code
- Libraries: NumPy

## THEORY

**Perceptron:** A perceptron is the simplest artificial neural network model inspired by a biological neuron. It takes weighted inputs, adds a bias, and passes the result through an activation function to produce an output. It is mainly used for binary classification problems.

**Single Layer Perceptron (SLP):** A Single Layer Perceptron consists of an input layer directly connected to an output layer without any hidden layers. It can solve only linearly separable problems such as AND and OR logic gates. Learning is performed using a simple weight update rule.

**Multi-Layer Perceptron (MLP):** A Multi-Layer Perceptron contains one or more hidden layers between the input and output layers. It uses nonlinear activation functions and is trained using the backpropagation algorithm. MLPs are capable of solving complex, non-linearly separable problems such as XOR.

**Epoch:** An epoch represents one complete pass of the entire training dataset through the neural network. During each epoch, the network updates its weights to reduce the error. Multiple epochs are usually required for effective learning.

**Learning Rate:** The learning rate controls the magnitude of weight updates during training. A smaller learning rate results in slow but stable learning, while a larger learning rate speeds up training but may cause instability. It is an important hyperparameter in neural networks.

**Weights:** Weights are numerical parameters associated with each input that determine its influence on the output. During training, weights are adjusted to minimize the error between predicted and actual outputs. Proper weight tuning improves model accuracy.

**Bias:** Bias is an additional parameter added to the weighted sum of inputs. It allows the activation function to be shifted, enabling the model to fit the data more flexibly. Bias helps the decision boundary avoid passing through the origin.

**Activation Function:** An activation function introduces non-linearity into the neural network and determines whether a neuron should be activated. Common activation functions include the step function, sigmoid function, and ReLU. Non-linearity allows networks to learn complex patterns.

**Linearly Separable Problem:** A problem is said to be linearly separable if a single straight line (or hyperplane) can separate the data points into different classes. Examples include AND and OR logic gates. Single Layer Perceptrons can solve such problems.

**Non-Linearly Separable Problem:** A non-linearly separable problem cannot be separated using a single straight line. The XOR problem is a classic example. Solving such problems requires hidden layers and nonlinear activation functions.

**Backpropagation:** Backpropagation is a supervised learning algorithm used in Multi-Layer Perceptrons. It calculates the error at the output layer and propagates it backward to update the weights. This process enables the network to learn complex relationships.

**Training Data:** Training data consists of input-output pairs used to train the neural network. The model learns by adjusting its parameters based on this data. High-quality and sufficient training data leads to better performance.

## IMPLEMENTATION

```
1 # Implement a perceptron to learn the AND logic gate.
2 # ( AND gate is Linearly separable and Perfect for single
3 # layer perceptron)
4
5 import numpy as np
6
7 # Training data for AND gate
8 X = np.array([
9     [0, 0],
10    [0, 1],
11    [1, 0],
12    [1, 1]
13 ])
14 y = np.array([0, 0, 0, 1]) # AND output
15
16 # Initialize weights and bias
17 weights = np.zeros(2)
18 bias = 0
19 learning_rate = 0.1
20
21 # Activation function
```

```

22 def step_function(x):
23     return 1 if x >= 0 else 0
24
25 # Training
26 for epoch in range(10):
27     for i in range(len(X)):
28         linear_output = np.dot(X[i], weights) + bias
29         y_pred = step_function(linear_output)
30         error = y[i] - y_pred
31
32     # Update rule
33     weights += learning_rate * error * X[i]
34     bias += learning_rate * error
35
36 print("Trained weights:", weights)
37 print("Trained bias:", bias)
38
39 # Testing
40 print("\nTesting AND gate:")
41 for i in range(len(X)):
42     output = step_function(np.dot(X[i], weights) + bias)
43     print(X[i], "->", output)

```

Listing 1: Single Layer Perceptron Implementation

```

1 Trained weights: [0.2 0.1]
2 Trained bias: -0.2000000000000004
3
4 Testing AND gate:
5 [0 0] -> 0
6 [0 1] -> 0
7 [1 0] -> 0
8 [1 1] -> 1

```

Listing 2: Output of the Single Layer Perceptron Implementation

```

1 # Implement an MLP to learn the XOR logic gate.
2 # (XOR gate is for Not linearly separable and it requires
3 # hidden layer)
4
5 import numpy as np
6
7 # XOR dataset
8 X = np.array([
9     [0, 0],
10    [0, 1],
11    [1, 0],
12    [1, 1]
13])
14 y = np.array([[0], [1], [1], [0]])

```

```

14
15 # Initialize weights
16 np.random.seed(1)
17 W1 = np.random.randn(2, 2)
18 b1 = np.zeros((1, 2))
19 W2 = np.random.randn(2, 1)
20 b2 = np.zeros((1, 1))
21
22 learning_rate = 0.1
23
24 # Activation functions
25 def sigmoid(x):
26     return 1 / (1 + np.exp(-x))
27
28 def sigmoid_derivative(x):
29     return x * (1 - x)
30
31 # Training
32 for epoch in range(5000):
33     # Forward pass
34     hidden_input = np.dot(X, W1) + b1
35     hidden_output = sigmoid(hidden_input)
36
37     output_input = np.dot(hidden_output, W2) + b2
38     y_pred = sigmoid(output_input)
39
40     # Backpropagation
41     error = y - y_pred
42     d_output = error * sigmoid_derivative(y_pred)
43
44     d_hidden = d_output.dot(W2.T) * sigmoid_derivative(
45         hidden_output)
46
47     # Update weights
48     W2 += hidden_output.T.dot(d_output) * learning_rate
49     b2 += np.sum(d_output, axis=0, keepdims=True) *
50         learning_rate
51     W1 += X.T.dot(d_hidden) * learning_rate
52     b1 += np.sum(d_hidden, axis=0, keepdims=True) *
53         learning_rate
54
55 # Testing
56 print("\nTesting XOR gate:")
57 for i in range(len(X)):
58     hidden = sigmoid(np.dot(X[i], W1) + b1)
59     output = sigmoid(np.dot(hidden, W2) + b2)
60     print(X[i], "->", round(output.item()))

```

---

Listing 3: Multi Layer Perceptron Implementation

```
1 Testing XOR gate:  
2 [0 0] -> 0  
3 [0 1] -> 1  
4 [1 0] -> 1  
5 [1 1] -> 0
```

Listing 4: Output of the Multi Layer Perceptron Implementation

## CONCLUSION

In this lab, we explored the concepts of Single Layer Perceptron (SLP) and Multi-Layer Perceptron (MLP). We implemented an SLP to learn the AND logic gate, which is linearly separable, and successfully trained it to produce correct outputs. We also implemented an MLP to learn the XOR logic gate, which is not linearly separable, and demonstrated its ability to classify the inputs correctly after training. These implementations highlight the capabilities of perceptrons in solving different types of classification problems.