

| **Title: To implement XOR LOGIC using perceptron network.** . |
| --- |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Objective:** Implement multilayer perceptron for XOR function with binary inputs and target.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

CO2 : Analyze various neural network architectures **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

* .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**Pre Lab/ Prior Concepts:**

**Perceptron Model**

A perceptron is a simple artificial neural network model that can be used for binary classification tasks. It consists of a single neuron with multiple inputs, each weighted, and a bias term. The weighted sum of the inputs is passed through an activation function (often a step function) to produce an output of 1 or 0.

**Linear separability**

A dataset is said to be linearly separable if it can be divided into two classes by a hyperplane. In the case of two-dimensional data, a hyperplane is a line. The perceptron is capable of learning to classify linearly separable data.

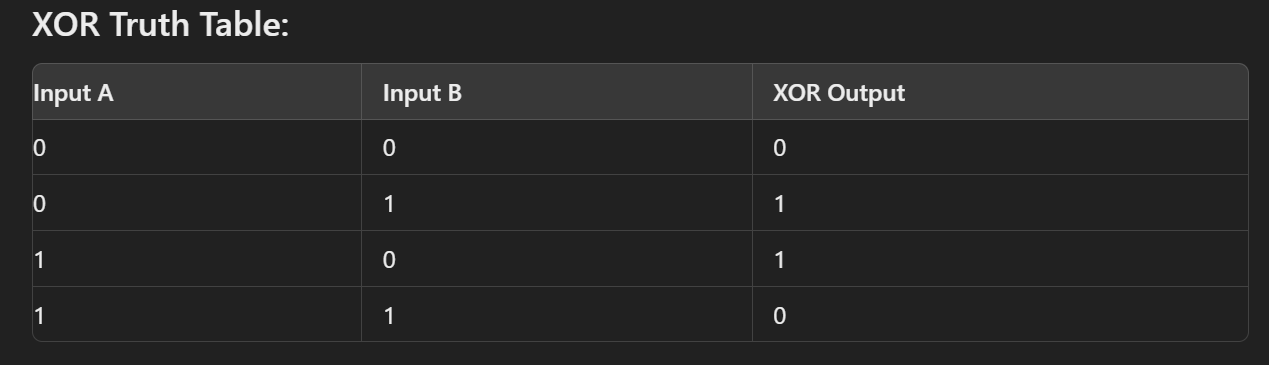
**Design of Classifier using Multi-layer Perceptron model for XOR logic of 2 inputs**

The XOR (exclusive OR) function is not linearly separable. This means a single-layer perceptron cannot learn to classify XOR data. However, a multi-layer perceptron (MLP) with at least one hidden layer can be used to solve this problem.

Here's a step-by-step approach to designing an MLP for XOR logic:

1. **Define the network architecture:**
   * **Input layer:** 2 neurons (for the two inputs)
   * **Hidden layer:** 1 neuron (sufficient for XOR)
   * **Output layer:** 1 neuron (for the binary output)
2. **Initialize weights and biases:** Assign random values to the weights and biases of the network.
3. **Forward propagation:**
   * Present an input pattern to the network.
   * Calculate the weighted sum of the inputs for each neuron in the hidden layer.
   * Apply the activation function (e.g., sigmoid) to the weighted sum to get the neuron's output.
   * Pass the outputs of the hidden layer neurons to the output layer and repeat the process to get the final output.
4. **Backpropagation:**
   * Calculate the error between the desired output and the actual output.
   * Use the chain rule to propagate the error back through the network and update the weights and biases to minimize the error.
5. **Repeat steps 3 and 4:** Continue training the network with multiple input patterns until the error reaches an acceptable level.

**Truth table for XOR logic**



**Classifier using Multi-layer Perception Network for implementing XOR logic for 2 inputs.**

**Code:**

# @title MLP Perceptron Learning

import numpy as np

# Sigmoid activation function

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Derivative of sigmoid function

def sigmoid\_derivative(x):

return x \* (1 - x)

# Training data for XOR

X = np.array([[0, 0],

[0, 1],

[1, 0],

[1, 1]])

# XOR outputs

y = np.array([[0], [1], [1], [0]])

# Set seed for reproducibility

np.random.seed(42)

# Initialize weights and biases for the MLP

inputLayer\_neurons = 2 # Number of input neurons (x1 and x2)

hiddenLayer\_neurons = 2 # Number of neurons in the hidden layer

output\_neurons = 1 # Number of output neurons (1 for XOR output)

# Random initialization of weights and biases

hidden\_weights = np.random.uniform(size=(inputLayer\_neurons, hiddenLayer\_neurons))

hidden\_bias = np.random.uniform(size=(1, hiddenLayer\_neurons))

output\_weights = np.random.uniform(size=(hiddenLayer\_neurons, output\_neurons))

output\_bias = np.random.uniform(size=(1, output\_neurons))

# Learning rate

learning\_rate = 0.5

# Training loop for the MLP

epochs = 10000

for epoch in range(epochs):

# Forward propagation

# Compute hidden layer activation

hidden\_layer\_activation = np.dot(X, hidden\_weights) + hidden\_bias

hidden\_layer\_output = sigmoid(hidden\_layer\_activation)

# Compute output layer activation

output\_layer\_activation = np.dot(hidden\_layer\_output, output\_weights) + output\_bias

predicted\_output = sigmoid(output\_layer\_activation)

# Backpropagation

# Calculate the error

error = y - predicted\_output

# Derivative of output (error \* derivative of sigmoid)

d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

# Error of hidden layer

error\_hidden\_layer = d\_predicted\_output.dot(output\_weights.T)

# Derivative of hidden layer output

d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

# Update the weights and biases

output\_weights += hidden\_layer\_output.T.dot(d\_predicted\_output) \* learning\_rate

output\_bias += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* learning\_rate

hidden\_weights += X.T.dot(d\_hidden\_layer) \* learning\_rate

hidden\_bias += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* learning\_rate

# Print the final predicted output after training

print("Final predicted output after training:")

print(np.round(predicted\_output, 4))

# Testing the MLP on XOR inputs

print("Testing the MLP on XOR inputs:")

for i in range(len(X)):

hidden\_layer\_activation = np.dot(X[i], hidden\_weights) + hidden\_bias

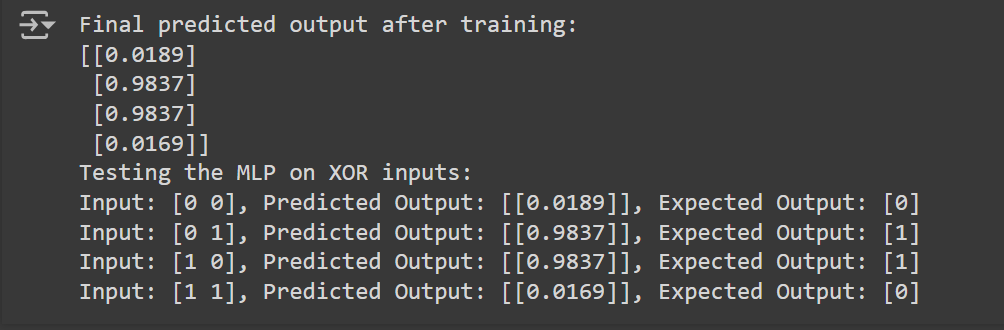
hidden\_layer\_output = sigmoid(hidden\_layer\_activation)

output\_layer\_activation = np.dot(hidden\_layer\_output, output\_weights) + output\_bias

predicted\_output = sigmoid(output\_layer\_activation)

print(f"Input: {X[i]}, Predicted Output: {np.round(predicted\_output, 4)}, Expected Output: {y[i]}")

**Output:**

****

**Conclusion:** Thus, we have successfully implemented classifier using MLP for linearly non-separable XOR functions of 2 inputs

**Post Lab Descriptive Questions :**

1. Why is XOR-logic function being Linearly not separable?

The XOR (exclusive OR) function is linearly non-separable because a linear classifier (such as a single-layer perceptron) cannot find a single straight line (or hyperplane in higher dimensions) to separate the classes of the XOR function in the input space.

Here's why XOR is non-separable by a single linear boundary:

XOR takes two binary inputs and returns true if and only if one of the inputs is true and the other is false.

The truth table for XOR is:



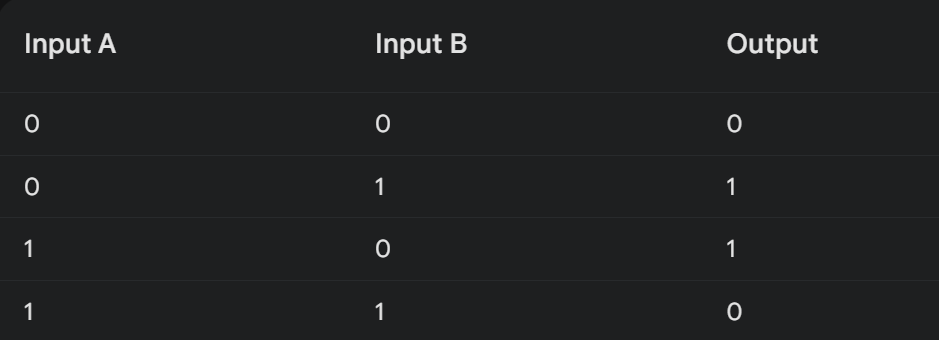
When you plot the inputs in 2D space, you get:

* (0, 0) → 0 (Class 0)
* (0, 1) → 1 (Class 1)
* (1, 0) → 1 (Class 1)
* (1, 1) → 0 (Class 0)

In this plot, there's no way to draw a single straight line to separate the two classes (Class 0 and Class 1). The points that need to be classified as '1' are on opposite corners of the space (i.e., (0, 1) and (1, 0)), and the points that need to be classified as '0' are also on opposite corners (i.e., (0, 0) and (1, 1)).

1. Design a classifier using MLP perceptron model for implementing NOT XOR logic

NOT XOR logic is a boolean function that outputs 1 only when the inputs are different. Here's its truth table:



To implement NOT XOR logic using an MLP perceptron, we'll use a single hidden layer with two neurons and an output layer with one neuron. This structure is sufficient to capture the non-linearity inherent in the XOR function. initialize the weights and biases randomly.

import tensorflow as tf

import numpy as np

# Create the model

model = tf.keras.Sequential([

tf.keras.layers.Dense(2, activation='sigmoid', input\_shape=(2,)),

tf.keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Create the dataset

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

y = [0, 1, 1, 0]

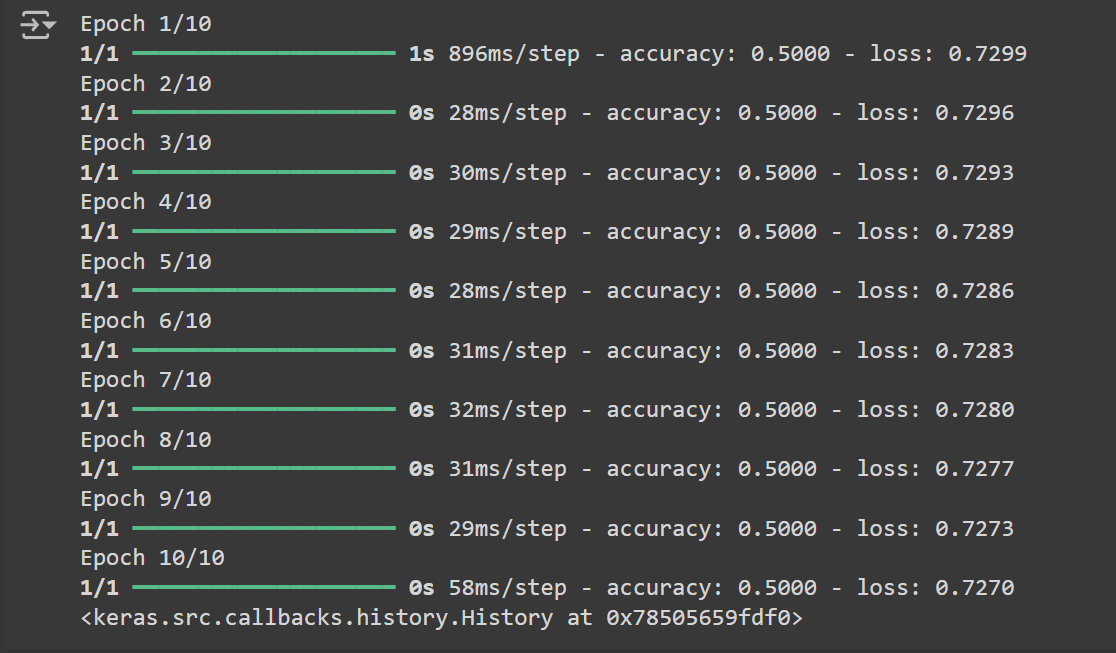
# Convert X and y to NumPy arrays

X = np.array(X)

y = np.array(y)

# Train the model

model.fit(X, y, epochs=10)



.

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of faculty in-charge**