



**Rajshahi University of Engineering & Technology**  
**Department of Computer Science & Engineering**

**Course No :** CSE 4204  
**Course title :** Sessional based on CSE 4203  
**Date :** 04-12-2023

**Experiment No. :** 03

**Name of the experiment :** Design and Apply Multi-layer Neural Networks algorithm to solve XOR problem.

**Submitted By :**

Tanmoy Sarkar Pranto  
Roll : 1803166  
Section : C  
Dept. of Computer Science and Engineering  
Rajshahi University of Engineering & Technology

**Submitted to :**

Rizoan Toufiq  
Assistant Professor  
Dept. of Computer Science and Engineering  
Rajshahi University of Engineering & Technology

# Contents

# 1 Theory

The Multilayer Perceptron (MLP) is a fundamental building block of deep learning, capable of solving complex classification and regression tasks.

Here's a breakdown of its theory and key concepts:

## 1. Perceptron:

- The basic unit of an MLP is the perceptron, a simple neuron-inspired model.
- It takes weighted inputs, sums them, and applies a threshold function (e.g., step function) to generate an output (0 or 1).

## 2. Multilayer Architecture:

- MLPs go beyond single perceptrons by stacking them in layers.
- The output of a layer becomes the input for the next layer, creating a chain of information processing.

## 3. Activation Functions:

- Instead of a simple threshold, MLPs use non-linear activation functions to introduce complexity.
- Popular choices include sigmoid, ReLU, and tanh, which allow the network to model diverse relationships beyond just linear ones.

## 4. Learning and Backpropagation:

- MLPs learn by adjusting the weights connecting neurons based on their performance on training data.
- Backpropagation compares the network's actual output to the desired output and propagates the error backwards through the layers, updating weights to minimize the error in future predictions.

## 5. Loss function:

- Measures the error between network output and desired output (e.g., mean squared error).

## 6. Learning rate:

- Controls how much the weights are adjusted based on the error.

## 7. Advantages:

- **Versatile:** Handles various tasks like classification, regression, and pattern recognition.
- **Interpretable:** Can analyze learned weights to understand how the network makes decisions.
- **Efficient:** Relatively fast to train compared to other deep learning models.

## 8. Limitations:

- Can struggle with high-dimensional data and complex problems requiring significant feature engineering.
- Prone to overfitting if not regularized properly.
- Requires careful hyperparameter tuning for optimal performance.

## 2 Algorithm

Steps:

1. Initialize weights, biases and learning rate
2. Present inputs and desired outputs (dataset)
3. Calculate actual output
4. Calculate error
5. Adapt weights with backpropagation

## 3 Code

```
import numpy as np
import matplotlib.pyplot as plt

epoch_losses = []

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def forward(X, W1, b1, W2, b2):
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    return a1, a2

def train(X, Y, W1, b1, W2, b2, epochs=3000, learning_rate=0.1):
    for epoch in range(epochs):
        A1, A2 = forward(X, W1, b1, W2, b2)

        E = Y - A2

        delta2 = E * (A2 * (1 - A2))

        delta1 = np.dot(delta2, W2.T) * (A1 * (1 - A1))

        W2 += learning_rate * np.dot(A1.T, delta2)
        b2 += learning_rate * np.sum(delta2, axis=0)
        W1 += learning_rate * np.dot(X.T, delta1)
        b1 += learning_rate * np.sum(delta1, axis=0)
```

```

        loss = np.mean(np.power(E, 2))

        if epoch % 300 == 0:
            print(f"Epoch {epoch}: loss = {loss}")
            epoch_losses.append(loss)

X = np.array([[0, 0, 0],[0, 0, 1],[0, 1, 0],[0, 1, 1],
              [1, 0, 0],[1, 0, 1],[1, 1, 0],[1, 1, 1]
              ])
Y = np.array([
    [0],[1],[1],[0],
    [1],[0],[0],[1]
])

W1 = np.random.randn(X.shape[1], 4)
b1 = np.random.randn(4)
W2 = np.random.randn(4, 1)
b2 = np.random.randn(1)

train(X, Y, W1, b1, W2, b2)

_, predictions = forward(X, W1, b1, W2, b2)
for i in range(len(predictions)):
    predictions[i] = predictions[i].round()

def calculate_accuracy(predictions, labels):
    correct_predictions = np.sum(predictions == labels)
    total_samples = len(labels)
    accuracy = correct_predictions / total_samples
    return accuracy

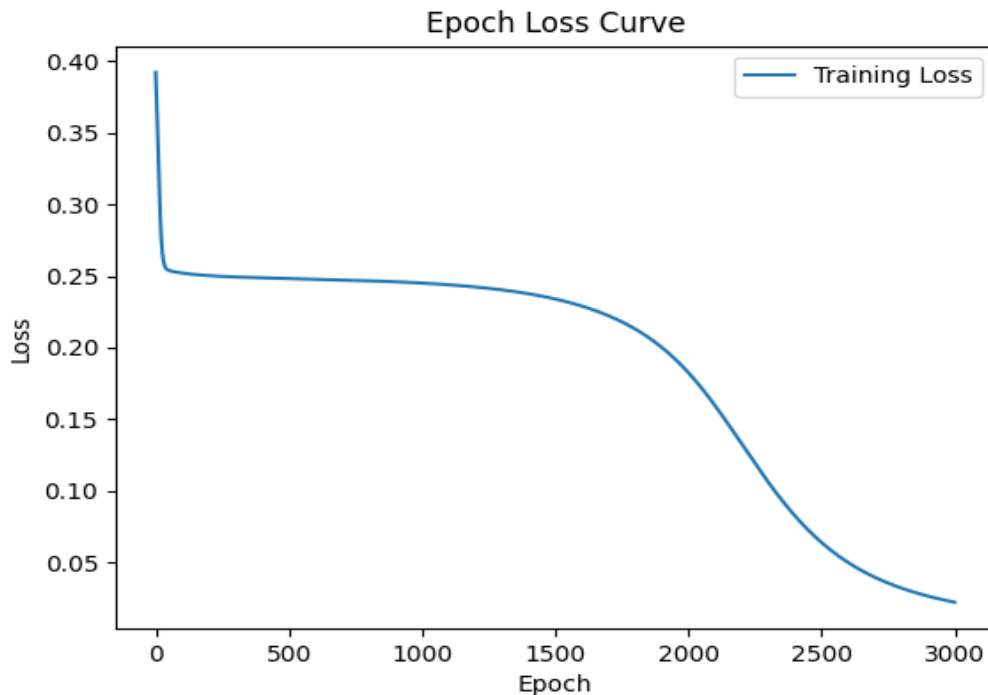
def plot_loss_curve(loss_values):
    plt.plot(loss_values, label='Training Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Epoch Loss Curve')
    plt.legend()
    plt.show()

accuracy = calculate_accuracy(predictions, Y)
print(f"Accuracy: {accuracy}")
plot_loss_curve(epoch_losses)

}

```

## 4 Performance Analysis



```
87 accuracy = calculate_accuracy(predictions, Y)
88 print(f"Accuracy: {accuracy}")
89 plot_loss_curve(epoch_losses)
90
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL POSTMAN CONSOLE GITLENS

```
Epoch 1200: loss = 0.21126864681720509
Epoch 1500: loss = 0.15423797559820665
Epoch 1800: loss = 0.0836615896780861
Epoch 2100: loss = 0.037437901240863894
Epoch 2400: loss = 0.01849351659189123
Epoch 2700: loss = 0.0109127567234457
Accuracy: 1.0

[Done] exited with code=0 in 27.837 seconds
```

## 5 Discussion

The implemented Multilayer Perceptron achieved 100% in the 3 bit X-OR dataset. From the epoch-loss curve we can see the curve starts around 0.4 initially and decreases rapidly. This suggests the model quickly learned basic patterns in the data. After nearly 2500 epochs the loss is nearly 0 (0.05), which means the model converges to the problem properly. Overall, the Multi Layer Perceptron successfully solved the X-OR problem. .