Neural Network From Scratch: Solving the XOR Problem

Objective: The goal of this project was to implement a simple 2-layer neural network using only the NumPy library to solve the classic XOR classification problem. The network was trained to correctly classify inputs by adjusting its weights and biases through backpropagation.

---

1. XOR Problem Setup

The XOR function is a fundamental logical operation that returns true (1) if the inputs are different, and false (0) if they are the same. The dataset was defined as follows:

- **Inputs (X):** A 4×2 matrix representing the four possible input pairs.
    - [0, 0]
    - [0, 1]
    - [1, 0]
    - [1, 1]
- **Outputs (Y):** A 4×1 matrix of the corresponding XOR results.
    - [0]
    - [1]
    - [1]
    - [0]

A visual representation of this data shows that it is **not linearly separable**, meaning a single straight line cannot separate the two classes (0s and 1s). This is why a simple single-layer perceptron would fail, and a hidden layer is required.

---

2. Neural Network Implementation

The network was built with a hidden layer to handle the non-linear nature of the problem.

Network Architecture:

- **Input Layer:** 2 neurons (for the two inputs).
- **Hidden Layer:** 4 neurons.
- **Output Layer:** 1 neuron (for the single output).

Activation Function:

The **sigmoid** function was used for both the hidden and output layers. This function squashes values into the range [0, 1], which is ideal for binary classification tasks.

Forward Propagation:

The forward pass calculates the network's prediction. For each layer, it involves a linear transformation followed by the activation function.

- **Hidden Layer:** A1=sigmoid(X·W1+b1)
- **Output Layer:** A2=sigmoid(A1·W2+b2)

Loss Function:

The **Mean Squared Error (MSE)** was used to measure the difference between the network's predictions and the true outputs.

---

3. Backpropagation and Training

The network was trained using **gradient descent**, which relies on the **backpropagation** algorithm to compute the gradients of the loss with respect to the weights and biases.

Gradient Calculation:

The gradients were derived manually using the chain rule, allowing us to determine how to adjust each weight and bias to reduce the overall loss.

- **Output Layer Gradients:** $\delta_2 = (A_2 - Y) \odot \text{sigmoid_derivative}(A_2)$
- **Hidden Layer Gradients:** $\delta_1 = \delta_2 \cdot W_2^T \odot \text{sigmoid_derivative}(A_1)$

These gradients were then used to update the weights and biases: Wnew=Wold−learning_rate·∂W∂L

Training Process:

A training loop ran for 60,000 epochs. The loss was monitored throughout the process and was observed to decrease steadily, confirming that the network was learning effectively.

---

4. Evaluation and Results

After training, the model was tested on the original XOR inputs.

Final Predictions:

- **Inputs:** [[0, 0], [0, 1], [1, 0], [1, 1]]
- **Predicted Outputs:** [[0.], [1.], [1.], [0.]]
- **Expected Outputs:** [[0], [1], [1], [0]]

The predictions perfectly match the expected outputs, demonstrating 100% accuracy on the training set.

Summary and Learnings:

This project successfully demonstrated how a multi-layer neural network can solve a non-linear problem like XOR. The key takeaway is the importance of a **hidden layer** with a **non-linear activation function**. This architectural choice allows the network to learn a complex, non-linear decision

boundary, which is impossible with a single-layer model. The successful implementation of **forward propagation** and **backpropagation** from scratch highlights the fundamental principles of training a neural network.