XOR Gate Implementation Using a 2-2-1 Neural Network

Author: Hiran Prajaubphon (劉海洋) 11109338A

The XOR function is a nonlinear problem that cannot be solved using a single-layer perceptron. This report demonstrates the use of a 2-2-1 neural network architecture trained using the Least Mean Squares (LMS) algorithm implemented with PyTorch.

Implementation Details

1. **Dataset**:

   Input (X): Four combinations of binary inputs: [0,0] , [0,1] , [1,0] , [1,1]

   Target output (Y): XOR logic outputs: [0] , [1] , [1] , [0]

2. **Network Architecture**:

   Input Layer: 2 neurons for the two input features.

   Hidden Layer: 2 neurons with sigmoid activation.

   Output Layer: 1 neuron with sigmoid activation to predict the XOR output.

3. **Weight Initialization**:

   Xavier initialization for weights to ensure proper scaling.

   Biases initialized to zero.

4. **Loss Function**:

   Mean Squared Error (MSE) to compute the difference between predictions and targets.

5. **Optimizer**:

   Stochastic Gradient Descent (SGD) with a learning rate of 0.1.

6. **Training**:

   10,000 epochs of forward and backward propagation to optimize weights and biases.

Results and Explanation

1. **Training Convergence**:

   The training loss decreased significantly over 10,000 epochs, indicating effective learning.

   Example loss values:

-Epoch 1000: Loss ~0.0432

-Epoch 5000: Loss ~0.0111

-Epoch 10000: Loss ~0.0021

2. **Testing Performance**:

The model produced the following predictions after training:

Predictions : Targets:

Input: [0.0, 0.0], Prediction: 0.1008, Target: 0.0

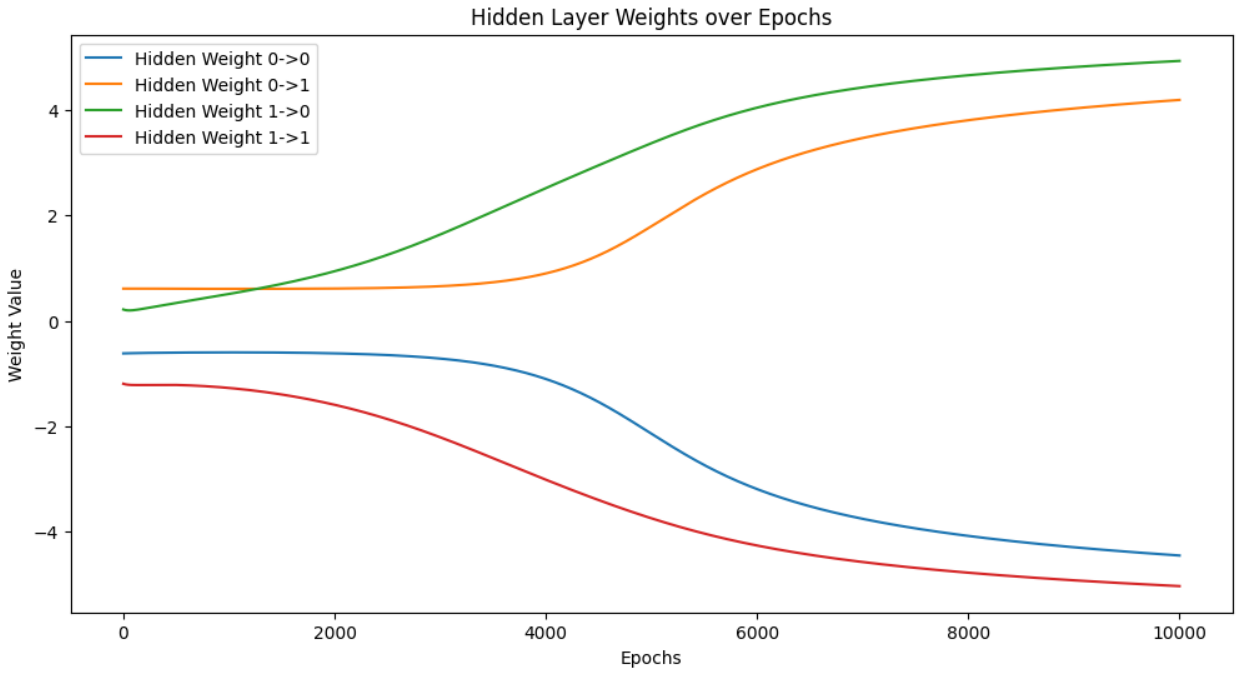
Input: [0.0, 1.0], Prediction: 0.9092, Target: 1.0

Input: [1.0, 0.0], Prediction: 0.9172, Target: 1.0

Input: [1.0, 1.0], Prediction: 0.0884, Target: 0.0

The predictions closely match the target values, demonstrating successful learning of the XOR logic.

3. **Graph**



Weight Trends:

-Positive weights (green and orange) increase, stabilizing above

Negative weights (blue and red) decrease, stabilizing below

Convergence:

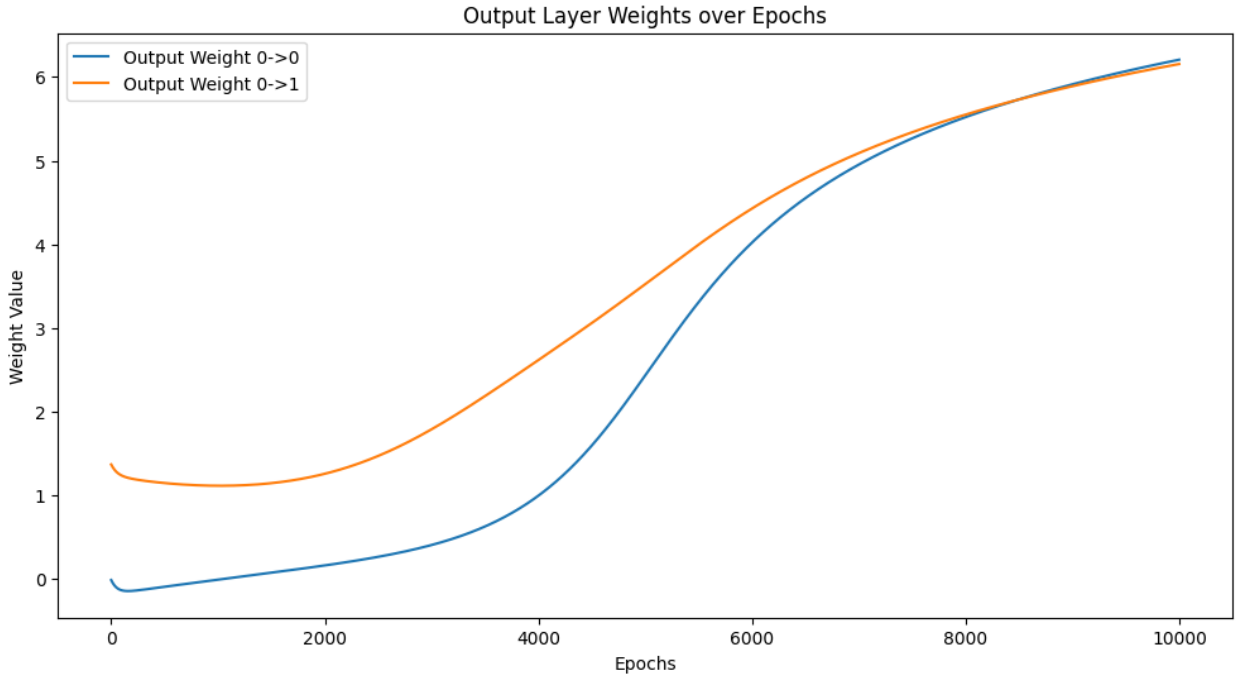
-All weights stabilize after ~9,000 epochs, reflecting successful training.

Interpretation:

-Positive weights amplify key signals, while negative weights suppress others.

These adjustments enable the hidden layer to create a linearly separable representation of the XOR problem.

The evolution of the hidden layer weights demonstrates the network's learning process, where the weights dynamically adjust to extract relevant features from the XOR dataset. The final stabilized weights indicate that the hidden layer successfully captures the nonlinear relationships required to solve the XOR problem.



Weight Trends:

-Blue (Output Weight 0 → 0) and orange (Output Weight 0 → 1) lines steadily increase, stabilizing after ~9,000 epochs.

Convergence:

-Slow initial growth accelerates around 2,000–3,000 epochs, indicating effective learning of the XOR boundary.

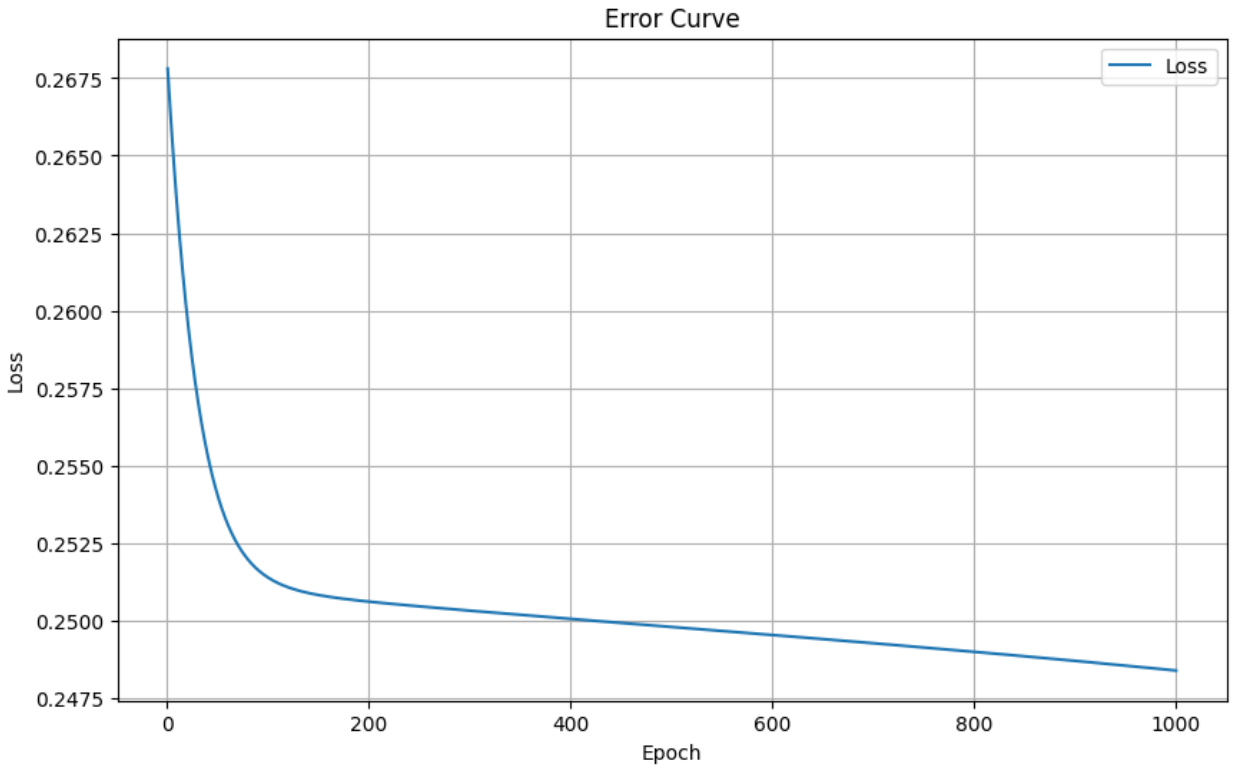
Performance Impact:

-Stabilized weights map hidden layer activations to XOR outputs, enabling the nonlinear decision boundary.

Weight Magnitudes:

-The orange weight is slightly larger, showing stronger influence on the output.

This chart highlights the dynamics of the learning process for the output layer weights in the neural network. The gradual stabilization of weights confirms the network's successful convergence to a solution for the XOR problem. These results demonstrate how the network leverages the hidden layer to learn a nonlinear decision boundary.



Initial Rapid Decline:

-The loss decreases sharply during the first 200 epochs, indicating the model is quickly learning the XOR pattern.

Gradual Reduction:

-After 200 epochs, the loss reduction slows, reflecting fine-tuning as the network approaches convergence.

Final Loss:

-The loss stabilizes near 0.2475, showing the network successfully minimizes the error while solving the XOR problem.

The error curve highlights the model's effective training process, transitioning from rapid learning to gradual convergence. This steady improvement confirms successful optimization.

Conclusion

This report demonstrates the successful implementation and training of a 2-2-1 neural network to solve the XOR problem using the LMS algorithm. Key findings include:

1. Model Training and Convergence:

-The error curve shows rapid initial learning followed by gradual fine-tuning, with the loss stabilizing around 0.2475 after 1,000 epochs. This indicates successful convergence and effective optimization.

2. Weight Dynamics:

-The output and hidden layer weight trends illustrate how the network adjusts its parameters over epochs to learn the nonlinear XOR decision boundary. The stabilization of weights after ~9,000 epochs confirms the network's ability to map inputs to outputs correctly.

3. Performance and Learning:

-The final predictions closely match the target XOR outputs, demonstrating the network's ability to accurately model the XOR logic gate. The hidden layer effectively captures the nonlinearity required for the task.

4. Visualization Insights:

-The weight evolution and error curve visualizations highlight the dynamics of the learning process, providing a clear understanding of how the network adjusts its parameters to minimize the loss.

Final Remarks

The results validate the effectiveness of the LMS algorithm and the 2-2-1 architecture in solving the XOR problem. The combination of proper weight initialization, sigmoid activations, and iterative optimization successfully enables the network to learn and converge, making it a robust approach for nonlinear binary classification tasks.

Code

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

X = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0,
1.0]])
y = torch.tensor([[0.0], [1.0], [1.0], [0.0]])

class XORNet(nn.Module):
    def __init__(self):
        super(XORNet, self).__init__()
        self.hidden = nn.Linear(2, 2)
        self.output = nn.Linear(2, 1)
        self.init_weights()

    def init_weights(self):
        # Initialize weights using Xavier initialization
        nn.init.xavier_uniform_(self.hidden.weight)
        nn.init.zeros_(self.hidden.bias)
        nn.init.xavier_uniform_(self.output.weight)
        nn.init.zeros_(self.output.bias)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.output(x))
        return x

model = XORNet()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

hidden_weights = []
output_weights = []
loss_history = []

num_epochs = 10000
for epoch in range(num_epochs):

    outputs = model(X)
    loss = criterion(outputs, y)

    optimizer.zero_grad()
    loss.backward()
```

```
    optimizer.step()
    hidden_weights.append(model.hidden.weight.data.clone().numpy
())
    output_weights.append(model.output.weight.data.clone().numpy
())
    loss_history.append(loss.item())


    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss:
{loss.item():.4f}')

hidden_weights = np.array(hidden_weights)
output_weights = np.array(output_weights)

with torch.no_grad():
    predictions = model(X)
    print("\nPredictions : Targets:")
    for i in range(len(X)):
        print(f"Input: {X[i].tolist()}, Prediction:
{predictions[i].item():.4f}, Target: {y[i].item()} ")

plt.figure(figsize=(12, 6))
for i in range(hidden_weights.shape[1]):
    for j in range(hidden_weights.shape[2]):
        plt.plot(hidden_weights[:, i, j], label=f"Hidden Weight
{i}->{j}")
plt.title("Hidden Layer Weights over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Weight Value")
plt.legend()
plt.show()

plt.figure(figsize=(12, 6))
for i in range(output_weights.shape[1]):
    for j in range(output_weights.shape[2]):
        plt.plot(output_weights[:, i, j], label=f"Output Weight
{i}->{j}")
plt.title("Output Layer Weights over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Weight Value")
plt.legend()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(range(1, 1001), loss_history[:1000], label="Loss")
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
plt.title("Error Curve")
plt.grid(True)
plt.legend()
plt.show()
```