UỶ BAN NHÂN DÂN THÀNH

PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC SÀI GÒN

◇◆◇

MÔN: XỬ LÝ NGÔN NGỮ TỰ NHIÊN

BÀI TẬP QUÁ TRÌNH

GIẢNG VIÊN HƯỚNG DẪN: PGS. TS. Nguyễn Tuấn Đăng

Nhóm:

Huỳnh Ngọc Mẫn – 3121411130

Nguyễn Hoàng Bảo Huy – 3121411086

Nguyễn Hữu Đức – 3121411058

Nguyễn Ngọc Kim Cương – 3121411032

Huỳnh Duy Khánh – 3121411099

TP.HỒ CHÍ MINH, NĂM 2024

Đề bài:

Thiết kế một mạng neuron đa lớp (Multilayer Perceptron - MLP) để giải quyết bài toán phân loại phi tuyến. Giả sử có bộ dữ liệu có hai đặc trưng x1, x2 và nhãn yE {0,1}. Hãy xây dựng một MLP với một hidden layer và một lớp đầu ra để phân loại dữ liệu.

Yêu cầu: cài đặt chương trình bằng Python.

Giải thích:

Bài toán XOR (Exclusive OR) là một bài toán phân loại nhị phân cơ bản trong máy học và thường được sử dụng để minh họa sự khác biệt giữa mô hình tuyến tính và phi tuyến tính, đồng thời cũng là một ví dụ kinh điển để giải thích lý do vì sao các mô hình nơ-ron đơn giản (như **Perceptron**) không thể giải quyết bài toán này mà cần đến mạng neuron đa lớp (**MLP**) hoặc các mô hình phức tạp hơn.

Dữ liệu mẫu:

- X = [(0,0), (0,1), (1,0), (1,1)]
- Y=[0, 1, 1, 0] (bài toán XOR)

Phép tính XOR của 2 bit cho giá trị 1 nếu đúng 1 trong 2 bit bằng 1 và cho giá trị bằng 0 trong các trường hợp còn lại.

x1	x2	x1 XOR x2
0	0	0
0	1	1
1	0	1
1	1	0

Các bước triển khai:

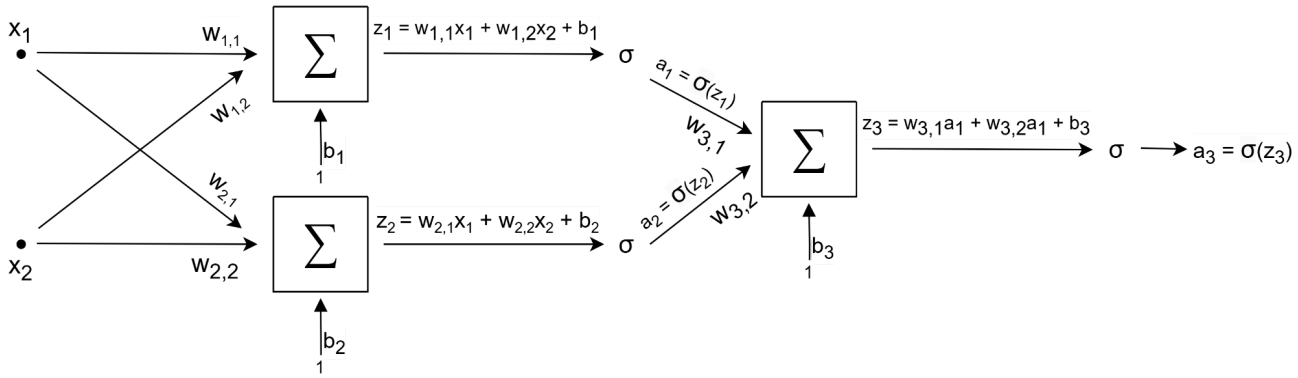
Bước 1: Kiến trúc mạng:

- Lớp đầu vào (Input layer): Mạng sẽ có 2 đầu vào là x1 và x2.
- Lớp ẩn (Hidden layer): Mạng sẽ có một lớp ẩn gồm 2 nơ-ron.
- Lớp đầu ra (Output layer): Mạng sẽ có 1 nơ-ron đầu ra, đưa ra dự đoán nhãn yyy, có giá trị 0 hoặc 1.

```python
# XOR inputs and outputs
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
Y = [0, 1, 1, 0]

# Define the structure of the neural network
input_layer_neurons = 2  # Number of features (x1, x2)
hidden_layer_neurons = 2  # Number of neurons in the hidden layer
output_neurons = 1  # Single output neuron
```

Mã khởi tạo đầu vào dữ liệu và kiến trúc mạng



Sơ đồ Kiến trúc mạng

Bước 2: Trọng số và hàm kích hoạt: Sử dụng hàm kích hoạt sigmoid cho cả lớp ẩn và lớp đầu ra.

Trọng số: Các trọng số sẽ được khởi tạo ngẫu nhiên (hoặc theo một phương pháp nhất định). Sử dụng random.uniform(-1, 1) để khởi tạo ngẫu nhiên các trọng số và bias trong khoảng từ -1 đến 1.

Tốc độ học: learning_rate = 0.1

Hàm kích hoạt: Sử dụng hàm kích hoạt Sigmoid cho cả lớp ẩn và lớp đầu ra. Hàm Sigmoid giúp biến đổi giá trị đầu ra về khoảng từ 0 đến 1.

Hàm sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$

Đạo hàm của hàm sigmoid: $\frac{d}{dx}\sigma(x) = \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right]$

```python
# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Randomly initialize weights and biases
weights_input_hidden = [[random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] for _ in range(input_layer_neurons)] # 2x2 matrix for input to hidden
weights_hidden_output = [random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] # 1x2 matrix for hidden to output
bias_hidden = [random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] # Bias for the hidden layer
bias_output = random.uniform(-1, 1) # Bias for the output layer
```

Mã khởi tạo các hàm kích hoạt và khởi tạo các trọng số ngẫu nhiên

Bước 3: Tính toán truyền tiến (forward pass):

- Tính toán đầu ra của lớp ẩn
  - Tính z_1 và z_2
    - $z_1 = w_{1,1}.x_1 + w_{1,2}.x_2 + b_1$
    - $z_2 = w_{2,1}.x_1 + w_{2,2}.x_2 + b_2$
  - Áp dụng hàm kích hoạt để tính a_1 và a_2

- $a_1 = \sigma(z_1) = \frac{1}{1 + e^{-z_1}}$
- $a_2 = \sigma(z_2) = \frac{1}{1 + e^{-z_2}}$

```python
# Forward pass
        x1, x2 = X[i]

        # Hidden layer calculations
        hidden_input = [
            x1 * weights_input_hidden[0][0] + x2 * weights_input_hidden[1][0] + bias_hidden[0],
            x1 * weights_input_hidden[0][1] + x2 * weights_input_hidden[1][1] + bias_hidden[1]
        ]
        hidden_output = [sigmoid(hidden_input[0]), sigmoid(hidden_input[1])]
```

Mã tính toán đầu ra của lớp ẩn

- Tính toán đầu ra của lớp đầu ra
  - Tính z_3
    - $z_3 = w_{3,1}.a_1 + w_{3,2}.a_2 + b_3$
  - Áp dụng hàm kích hoạt (Sigmoid) để tính a_3
    - $a_3 = \sigma(z_3) = \frac{1}{1 + e^{-z_3}}$

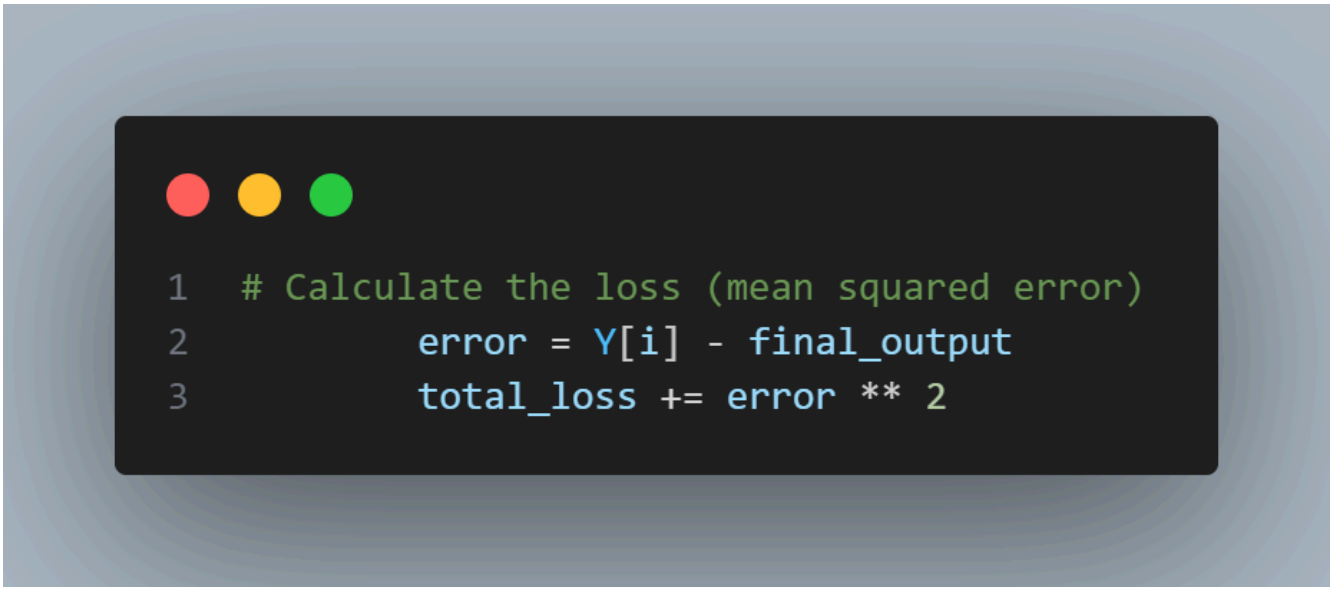
```python
# Output layer calculations
        final_input = hidden_output[0] * weights_hidden_output[0] + hidden_output[1] * weights_hidden_output[1] + bias_output
        final_output = sigmoid(final_input)
```

Mã tính toán đầu ra của lớp đầu ra

Bước 4: Backpropagation và cập nhật trọng số: Sử dụng backpropagation để

tính sai số và điều chỉnh trọng số của tất cả các lớp dựa trên quy tắc gradient descent.

- Tính sai số cho mỗi mẫu dữ liệu (Error Calculation)
  - $error = y - \hat{y}$
- Tính hàm mất mát (Loss)
  - $Loss = \frac{1}{2}(y - \hat{y})$



```
# Calculate the loss (mean squared error)
        error = Y[i] - final_output
        total_loss += error ** 2
```

Mã tính toán sai số giữa đầu ra dự đoán và thực tế

- Lan truyền ngược (Backpropagation)
  - Tính gradient của hàm mất mát đối với đầu ra của mạng, sử dụng đạo hàm của hàm kích hoạt (sigmoid)
    - $\delta_{output} = error \cdot \sigma'(\hat{y})$
  - Tính gradient của sai số đối với mỗi neuron trong lớp ẩn
    - $\delta_{hidden_1} = \delta_{output} \cdot w_{3,1} \cdot \sigma'(a_1)$
    - $\delta_{hidden_2} = \delta_{output} \cdot w_{3,2} \cdot \sigma'(a_2)$

```python
# Backpropagation
        # Calculate gradients for the output layer
        d_output = error * sigmoid_derivative(final_output)

        # Calculate gradients for the hidden layer
        d_hidden = [
            d_output * weights_hidden_output[0] * sigmoid_derivative(hidden_output[0]),
            d_output * weights_hidden_output[1] * sigmoid_derivative(hidden_output[1])
        ]
```

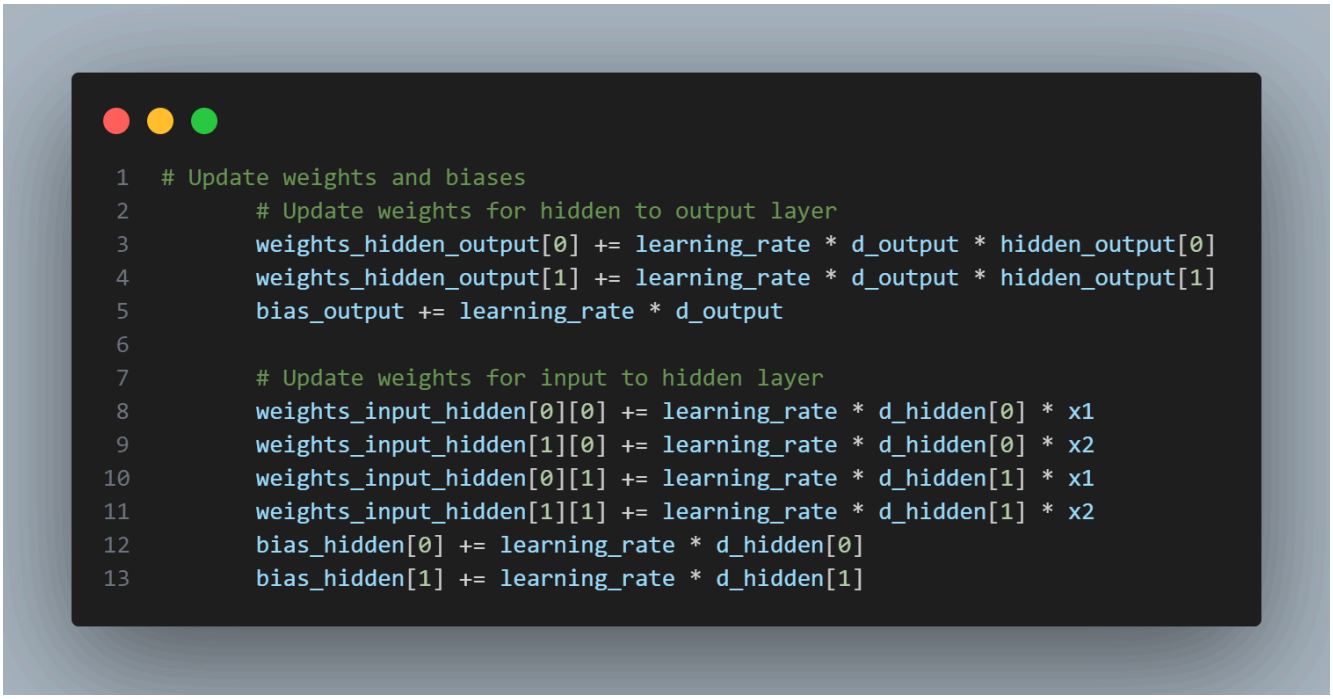
Mã tính toán gradient của lớp ẩn và lớp đầu ra

- Cập nhật trọng số từ lớp ẩn đến lớp đầu ra
  - Cập nhật các trọng số liên kết giữa lớp ẩn và lớp đầu ra
    - $w_{3,1} = w_{3,1} + \alpha \,.\, \delta_{output} \,.\, a_1$
    - $w_{3,2} = w_{3,2} + \alpha \,.\, \delta_{output} \,.\, a_2$
  - Cập nhật bias của lớp đầu ra
    - $b_3 = b_3 + \alpha \,.\, \delta_{output}$
- Cập nhật trọng số từ lớp đầu vào đến lớp ẩn
  - Cập nhật các trọng số liên kết giữa lớp đầu vào và lớp ẩn
    - $w_{1,1} = w_{3,1} + \alpha \,.\, \delta_{hidden_1} \,.\, x_1$
    - $w_{2,1} = w_{2,1} + \alpha \,.\, \delta_{hidden_1} \,.\, x_2$
    - $w_{1,2} = w_{1,2} + \alpha \,.\, \delta_{hidden_2} \,.\, x_1$
    - $w_{2,2} = w_{2,2} + \alpha \,.\, \delta_{hidden_2} \,.\, x_2$
  - Cập nhật bias của lớp ẩn
    - $b_1 = b_1 + \alpha \,.\, \delta_{hidden_1}$
    - $b_2 = b_2 + \alpha \,.\, \delta_{hidden_2}$

Trong đó:

α là tốc độ học (learning rate).

σ′ là đạo hàm của hàm kích hoạt (ở đây là sigmoid).



```
# Update weights and biases
        # Update weights for hidden to output layer
        weights_hidden_output[0] += learning_rate * d_output * hidden_output[0]
        weights_hidden_output[1] += learning_rate * d_output * hidden_output[1]
        bias_output += learning_rate * d_output

        # Update weights for input to hidden layer
        weights_input_hidden[0][0] += learning_rate * d_hidden[0] * x1
        weights_input_hidden[1][0] += learning_rate * d_hidden[0] * x2
        weights_input_hidden[0][1] += learning_rate * d_hidden[1] * x1
        weights_input_hidden[1][1] += learning_rate * d_hidden[1] * x2
        bias_hidden[0] += learning_rate * d_hidden[0]
        bias_hidden[1] += learning_rate * d_hidden[1]
```

Mã cập nhật trọng số cho lớp ẩn đến lớp đầu ra và lớp đầu vào đến lớp ẩn

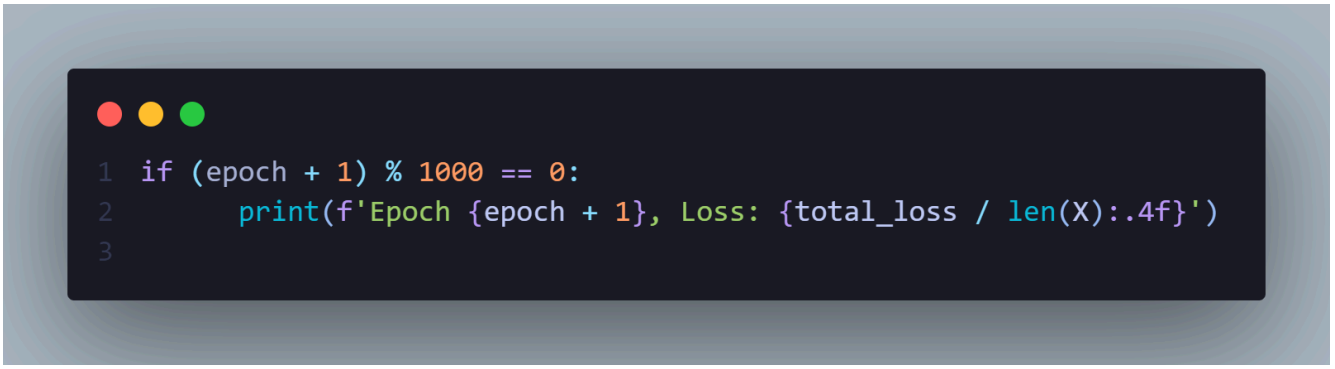
Bước 5: Huấn luyện mạng: Chạy qua nhiều epoch cho đến khi mạng có thể học được cách phân loại chính xác dữ liệu XOR.

Mạng được huấn luyện qua 10,000 epoch, với mục đích cho phép mạng có đủ thời gian học và tối ưu hóa trọng số để đạt độ chính xác cao.

```
# Training loop
epochs = 10000
for epoch in range(epochs):
```

Mã khởi tạo số và vòng lặp qua mỗi epoch

In lỗi trung bình sau mỗi 1000 epoch: Để theo dõi tiến độ huấn luyện, lỗi trung bình được in ra sau mỗi 1000 epoch, cho phép đánh giá quá trình hội tụ của mạng để theo dõi tiến trình huấn luyện.



```
if (epoch + 1) % 1000 == 0:
        print(f'Epoch {epoch + 1}, Loss: {total_loss / len(X):.4f}')

```

Mã in ra lỗi trung bình mỗi 1000 epoch

Kết quả:

Kết quả sau khi huấn luyện mạng thể hiện qua 10,000 epoch và giá trị hàm Loss giảm dần qua từng giai đoạn như bên dưới:

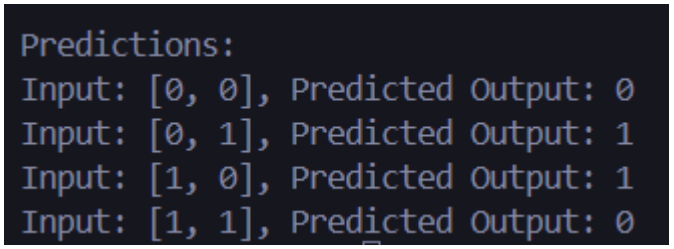
Epoch	Loss
1000	0.2499
2000	0.2315
3000	0.1480
4000	0.0278
5000	0.0113
6000	0.0067
7000	0.0047
8000	0.0036
9000	0.0029
10000	0.0024

Từ kết quả trên, ta thấy được giá trị hàm Loss giảm dần qua từng giai đoạn. Ở giai đoạn đầu tiên (1000 epoch) giá trị hàm Loss đạt 0.2499 một con số khá cao. Tuy nhiên con số này giảm xuống một cách đáng kể sau epoch 4000, giá trị hàm Loss chỉ còn 0.0278.

Kết quả này cho thấy mô hình đã dần học được các trọng số, cũng như là giảm độ lệch của kết quả, giúp nó giảm thiểu lỗi trong dự đoán và tiến tới hội tụ.

Đánh giá kết quả dự đoán:

Để đánh giá khả năng tính toán của mạng sau khi được huấn luyện, dựa vào dữ liệu mẫu, tiến hành cho dự đoán kết quả cho bài toán. Kết quả như sau:



Hình ảnh kết quả dự đoán

Từ kết quả dự đoán trên, ta thấy được mạng đã cho ra kết quả đúng theo tất cả mẫu liệu trong bài toán XOR.

Code tổng quát:

XLNNTN.py

```python
import math
import random

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# XOR inputs and outputs
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
Y = [0, 1, 1, 0]

# Define the structure of the neural network
input_layer_neurons = 2  # Number of features (x1, x2)
hidden_layer_neurons = 2  # Number of neurons in the hidden layer
output_neurons = 1  # Single output neuron

# Randomly initialize weights and biases
weights_input_hidden = [[random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] for _ in range(input_layer_neurons)] # 2x2 matrix for input to hidden
weights_hidden_output = [random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] # 1x2 matrix for hidden to output
bias_hidden = [random.uniform(-1, 1) for _ in range(hidden_layer_neurons)] # Bias for the hidden layer
bias_output = random.uniform(-1, 1) # Bias for the output layer

# Learning rate
learning_rate = 0.1
```

XLNNTN.py

```python
# Training loop
epochs = 10000
for epoch in range(epochs):
    total_loss = 0
    for i in range(len(X)):
        # Forward pass
        x1, x2 = X[i]

        # Hidden layer calculations
        hidden_input = [
            x1 * weights_input_hidden[0][0] + x2 * weights_input_hidden[1][0] + bias_hidden[0],
            x1 * weights_input_hidden[0][1] + x2 * weights_input_hidden[1][1] + bias_hidden[1]
        ]
        hidden_output = [sigmoid(hidden_input[0]), sigmoid(hidden_input[1])]

        # Output layer calculations
        final_input = hidden_output[0] * weights_hidden_output[0] + hidden_output[1] * weights_hidden_output[1] + bias_output
        final_output = sigmoid(final_input)

        # Calculate the loss (mean squared error)
        error = Y[i] - final_output
        total_loss += error ** 2

        # Backpropagation
        # Calculate gradients for the output layer
        d_output = error * sigmoid_derivative(final_output)

        # Calculate gradients for the hidden layer
        d_hidden = [
            d_output * weights_hidden_output[0] * sigmoid_derivative(hidden_output[0]),
            d_output * weights_hidden_output[1] * sigmoid_derivative(hidden_output[1])
        ]

        # Update weights and biases
        # Update weights for hidden to output layer
        weights_hidden_output[0] += learning_rate * d_output * hidden_output[0]
        weights_hidden_output[1] += learning_rate * d_output * hidden_output[1]
        bias_output += learning_rate * d_output

        # Update weights for input to hidden layer
        weights_input_hidden[0][0] += learning_rate * d_hidden[0] * x1
        weights_input_hidden[1][0] += learning_rate * d_hidden[0] * x2
        weights_input_hidden[0][1] += learning_rate * d_hidden[1] * x1
        weights_input_hidden[1][1] += learning_rate * d_hidden[1] * x2
        bias_hidden[0] += learning_rate * d_hidden[0]
        bias_hidden[1] += learning_rate * d_hidden[1]

    if (epoch + 1) % 1000 == 0:
        print(f'Epoch {epoch + 1}, Loss: {total_loss / len(X):.4f}')
```


XLNNTN.py

```python
# Testing the model
print("\nPredictions:")
for i in range(len(X)):
    x1, x2 = X[i]
    hidden_input = [
        x1 * weights_input_hidden[0][0] + x2 * weights_input_hidden[1][0] + bias_hidden[0],
        x1 * weights_input_hidden[0][1] + x2 * weights_input_hidden[1][1] + bias_hidden[1]
    ]
    hidden_output = [sigmoid(hidden_input[0]), sigmoid(hidden_input[1])]
    final_input = hidden_output[0] * weights_hidden_output[0] + hidden_output[1] * weights_hidden_output[1] + bias_output
    final_output = sigmoid(final_input)
    print(f"Input: {X[i]}, Predicted Output: {round(final_output)}")
```