Lab Assignment 1

Student Information:

- **Name:** Nirmal Chaturvedi
- **Roll Number:** 202201040210
- **Batch:** T4
- **Date of Submission:** 15/01/25

Neural Network Implementation from Scratch

Objective:

Implement a simple feedforward neural network from scratch in Python without using any in-built deep learning libraries. This implementation will focus on the fundamental components:

- Forward Pass
- Backpropagation
- Training using Gradient Descent

Problem Definition

Dataset:

- **Name:** XOR Dataset
- **Description:** The XOR dataset consists of four samples with two input features and one target value. It is commonly used as a benchmark to test the capability of neural networks to learn non-linear patterns.

Task:

- **Type:** Binary Classification
- **Objective:** The neural network should correctly classify the XOR dataset by predicting the target value for each input combination.

Methodology

Neural Network Architecture:

- **Input Layer:** 2 neurons (representing the input features).
- **Hidden Layer:** 4 neurons with Sigmoid activation function.
- **Output Layer:** 1 neuron with Sigmoid activation function (for binary classification).
- **Learning Rate:** 0.1 (used for gradient descent optimization).

Forward Pass:

The forward pass involves:

1. Computing the weighted sum of inputs and biases for the hidden layer.
2. Applying the activation function (Sigmoid) to the weighted sum.
3. Propagating the outputs of the hidden layer to the output layer.
4. Applying the activation function (Sigmoid) to the output layer's weighted sum to produce final predictions.

Backpropagation:

- The backpropagation algorithm minimizes the error by adjusting weights and biases based on the gradient of the loss function.
- **Steps:**
    1. Compute the error at the output layer.
    2. Calculate the gradient of the error with respect to weights and biases.
    3. Propagate the error backward through the network to adjust hidden layer weights.

Loss Function:

- **Type:** Mean Squared Error (MSE)
- **Formula:**

Optimization:

- **Method:** Gradient Descent
- Updates weights and biases using the formula:

Where is the learning rate.

Code Implementation

```
import numpy as np


class FeedForwardNN:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, lr=0.01):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.lr = lr


        # Initialize weights and biases
        self.w_input_hidden = np.random.randn(self.input_nodes,
self.hidden_nodes)
        self.b_hidden = np.zeros((1, self.hidden_nodes))
        self.w_hidden_output = np.random.randn(self.hidden_nodes,
self.output_nodes)
        self.b_output = np.zeros((1, self.output_nodes))


    def activation_function(self, x):
        return 1 / (1 + np.exp(-x))


    def activation_derivative(self, x):
        return x * (1 - x)


    def forward_propagation(self, inputs):
        self.hidden_input = np.dot(inputs, self.w_input_hidden) + self.b_hidden
        self.hidden_output = self.activation_function(self.hidden_input)
```

```
        self.final_input = np.dot(self.hidden_output, self.w_hidden_output) +
self.b_output
        self.final_output = self.activation_function(self.final_input)

        return self.final_output

    def backward_propagation(self, inputs, targets, predictions):
        output_error = targets - predictions
        output_gradient = output_error * self.activation_derivative(predictions)

        hidden_error = np.dot(output_gradient, self.w_hidden_output.T)
        hidden_gradient = hidden_error *
self.activation_derivative(self.hidden_output)

        self.w_hidden_output += np.dot(self.hidden_output.T, output_gradient) *
self.lr
        self.b_output += np.sum(output_gradient, axis=0, keepdims=True) * self.lr
        self.w_input_hidden += np.dot(inputs.T, hidden_gradient) * self.lr
        self.b_hidden += np.sum(hidden_gradient, axis=0, keepdims=True) *
self.lr

    def train(self, inputs, targets, epochs):
        for epoch in range(epochs):
            predictions = self.forward_propagation(inputs)
            self.backward_propagation(inputs, targets, predictions)
            if epoch % 1000 == 0:
                loss = np.mean((targets - predictions) ** 2)
                print(f"Epoch {epoch}: Loss = {loss}")
```

```
if __name__ == "__main__":
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([[0], [1], [1], [0]])

    input_nodes = 2
    hidden_nodes = 4
    output_nodes = 1
    learning_rate = 0.1

    nn = FeedForwardNN(input_nodes, hidden_nodes, output_nodes,
learning_rate)
    nn.train(inputs, targets, epochs=10000)

    print("Final Predictions:")
    print(nn.forward_propagation(inputs))
```

Screenshot-

```
    # Part C: Training the Network
    def train(self, inputs, targets, epochs):
        for epoch in range(epochs):
            predictions = self.forward_propagation(inputs)
            self.backward_propagation(inputs, targets, predictions)
            if epoch % 100 == 0:
                loss = np.mean((targets - predictions) ** 2)
                print(f"Epoch {epoch}: Loss = {loss}")

[23]: # Part C: Training the Network

[24]:
    def train(self, inputs, targets, epochs):
        for epoch in range(epochs):
            predictions = self.forward_propagation(inputs)
            self.backward_propagation(inputs, targets, predictions)
            if epoch % 100 == 0:
                loss = np.mean((targets - predictions) ** 2)
                print(f"Epoch {epoch}: Loss = {loss}")

[25]: #Part D: Results

[26]:
if __name__ == "__main__":
    # XOR dataset
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    targets = np.array([[0], [1], [1], [0]])

    # Neural network configuration
    input_nodes = 2
    hidden_nodes = 4
    output_nodes = 1
    learning_rate = 0.1

    # Create and train the neural network
    nn = FeedForwardNN(input_nodes, hidden_nodes, output_nodes, learning_rate)
    nn.train(inputs, targets, epochs=10000)

    # Display results
    print("Final Predictions:")
    print(nn.forward_propagation(inputs))
```

```
Epoch 7300: Loss = 0.0029609710662079677
Epoch 7400: Loss = 0.0028898756426998565
Epoch 7500: Loss = 0.0028217995554757737
Epoch 7600: Loss = 0.0027565629190460096
Epoch 7700: Loss = 0.0026939994606266626
Epoch 7800: Loss = 0.002633955282633539
Epoch 7900: Loss = 0.002576287755574727
Epoch 8000: Loss = 0.0025208645258074816
Epoch 8100: Loss = 0.0024675626246787328
Epoch 8200: Loss = 0.002416267667325817
Epoch 8300: Loss = 0.002366873130920267
Epoch 8400: Loss = 0.002319279703431458
Epoch 8500: Loss = 0.002273394695101845
Epoch 8600: Loss = 0.002229131505787076
Epoch 8700: Loss = 0.002186409142146447
Epoch 8800: Loss = 0.0021451517793901116
Epoch 8900: Loss = 0.0021052883629154393
Epoch 9000: Loss = 0.0020667522457100455
Epoch 9100: Loss = 0.002029480857873532
Epoch 9200: Loss = 0.001993415405025239
Epoch 9300: Loss = 0.0019585005927279186
Epoch 9400: Loss = 0.0019246843743755352
Epoch 9500: Loss = 0.0018919177202728803
Epoch 9600: Loss = 0.0018601544058801118
Epoch 9700: Loss = 0.0018293508174120278
Epoch 9800: Loss = 0.0017994657731728767
Epoch 9900: Loss = 0.00177046035917616
Final Predictions:
[[0.03945817]
 [0.95121886]
 [0.96815892]
 [0.04493095]]
```

```python
#  Importing Necessary Libraries
```

```python
import numpy as np
```

```python
#Part B: Implementation of the Neural Network
```

```python
class FeedForwardNN:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, lr=0.01):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.lr = lr

        # Initialize weights and biases
        self.w_input_hidden = np.random.randn(self.input_nodes, self.hidden_nodes)
        self.b_hidden = np.zeros((1, self.hidden_nodes))
        self.w_hidden_output = np.random.randn(self.hidden_nodes, self.output_nodes)
        self.b_output = np.zeros((1, self.output_nodes))

    def activation_function(self, x):
        return 1 / (1 + np.exp(-x))

    def activation_derivative(self, x):
        return x * (1 - x)

    def forward_propagation(self, inputs):
        # Calculate hidden layer activations
        self.hidden_input = np.dot(inputs, self.w_input_hidden) + self.b_hidden
        self.hidden_output = self.activation_function(self.hidden_input)

        # Calculate output layer activations
        self.final_input = np.dot(self.hidden_output, self.w_hidden_output) + self.b_output
        self.final_output = self.activation_function(self.final_input)

        return self.final_output

    def backward_propagation(self, inputs, targets, predictions):
        # Compute output layer error and gradients
        output_error = targets - predictions
        output_gradient = output_error * self.activation_derivative(predictions)
```

Declaration

I, Nirmal Chaturvedi, confirm that the work submitted in this assignment is my own and has been completed following academic integrity guidelines. The code is uploaded on my GitHub repository account, and the repository link is provided below:

- **GitHub Repository Link:-** https://github.com/nirmalchaturvedi/Deeplearning-Assignmnet-1

Signature: Nirmal Chaturvedi