CSE473s: Computational Intelligence – Fall 2025

Milestone 1 :

Name	Code
Sara Saber Samuel	2101138
Clara Ashraf Younan	2100932
Bishoy Tarek Soliman	2101067
Mina Ezzat Ragheb	2101363
Mark Matta Guirguis	2100372

1. Introduction

In this project, a custom neural network framework was implemented from scratch. The goal was to understand the internal workings of neural networks, including forward propagation, backward propagation, loss computation, and optimization. The implementation includes modular components such as layers, activation functions, loss functions, an optimizer, and a sequential model class. The system was tested on the XOR classification problem.

2. Library Design and Architecture Choices

The library is organized into modular files inside the 'lib' directory:

- activations.py: Contains Sigmoid and Tanh activation functions.

- layers.py: Contains the Dense (fully connected) layer.

- losses.py: Includes MSE (Mean Squared Error) and BCE (Binary Cross Entropy) loss functions.

- optimizer.py: Implements Stochastic Gradient Descent (SGD).

- network.py: Contains the Sequential class for chaining layers.

This modular design improves readability, maintainability, and reusability.

2.1 Layers (layers.py)

The Dense layer keeps two things: a weight matrix **W** and a bias vector **b**. We start these weights with small random values so the network can learn properly. In the forward pass, the layer calculates a simple equation: **input × W + b**. In the backward pass, it figures out how much each weight affected the error, saves these gradients, and updates the weights to improve the model.

2.2 Activations (activations.py)

Sigmoid and Tanh activation functions are implemented. Both store their outputs during forward passes and compute derivatives during backpropagation. Sigmoid outputs values between 0 and 1, useful for binary output. Tanh outputs between -1 and 1 and is used for hidden layers.

2.3 Loss Functions (losses.py)

The loss functions was implemented:

- MSE: Computes mean squared error, suitable for regression and simple tasks like XOR.

It provides forward (loss calculation) and backward (gradient) methods.

2.4 Optimizer (optimizer.py)

The SGD optimizer applies gradient descent updates to parameters using:

param = param - lr * gradient

This simple optimizer allows straightforward weight updates during backpropagation.

2.5 Sequential Model (network.py)

The Sequential class manages the complete forward and backward passing of data. Forward propagation applies each layer in order, and backward propagation reverses the order to compute gradients through the network.

3. XOR Test Results

The XOR problem was used as a benchmark test. A network with:

- Input layer: 2 neurons

- Hidden layer: 4 neurons + Tanh

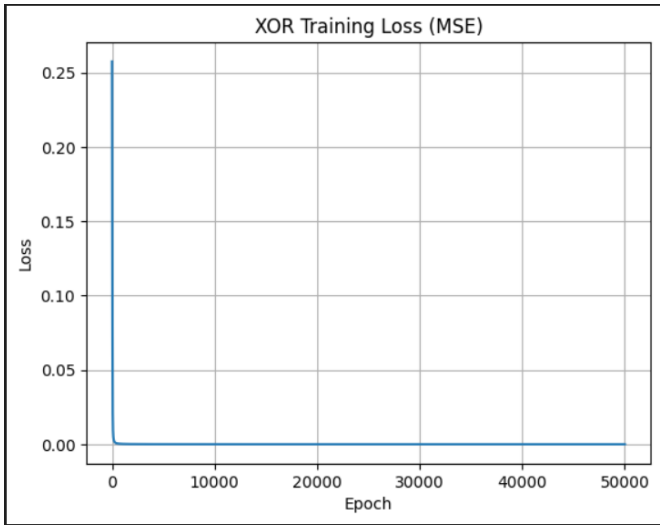
- Output layer: 1 neuron + Sigmoid

- Loss: MSE

- Optimizer: SGD

- Epochs: 50,000

was able to successfully learn the XOR function.



Final predictions closely matched the expected values, and the loss decreased smoothly, demonstrating correct learning behavior.

4. Challenges Faced

- Implementing correct backward propagation

- Designing modular library structure

- Ensuring correct gradient flow

- Implementing numerical gradient checking to validate gradients

5. Lessons Learned

Lessons learned from building the system:

- Importance of modular software design

- Sensitivity of training to learning rate and weight initialization

- Gradient checking is extremely useful

- Experience with debugging numerical ML systems