



## CSE473s - Computational Intelligence

### Neural Network Library & Advanced Applications

#### Team 6

NAME	ID
AHMED MOSTAFA ABDOU	2101004
AYMAN ABD AL-ELAH AL SHEBAH	2101760
AHMED SAYED MOHAMED	2101798

#### Submitted To:

- Dr. Hossam Hassan
- Eng. Abdallah Awdallah

## Contents

❖ Introduction:	1
➤ Project Overview .....	1
➤ Scope and Objectives .....	1
1. Library Design and Architecture Choices.....	2
1.1 Modular Architecture.....	2
1.2 Forward and Backward Propagation Flow.....	3
2. Results from the XOR Test .....	3
2.1 Experimental Setup.....	3
2.2 Training Results.....	4
2.3 Final Predictions .....	4
2.4 Conclusion of Validation.....	4



## ❖ Introduction:

### ➤ Project Overview

- In the domain of Computational Intelligence, understanding the mathematical underpinnings of deep learning is as critical as mastering high-level frameworks. This project focuses on the development of a foundational Neural Network library built entirely from scratch using only Python and NumPy.
- The primary objective is to move beyond the "black box" abstraction provided by modern libraries like TensorFlow. By manually implementing core components: including dense layers, activation functions, loss calculations, and optimization algorithms, we gain a rigorous understanding of the backpropagation algorithm and the flow of gradients through a computational graph.

### ➤ Scope and Objectives

- The project is structured into three main phases, designed to bridge the gap between theoretical concepts and practical application:
  1. **Library Implementation & Validation:** The initial phase involves constructing a modular library capable of defining sequential models. This includes implementing forward and backward passes for fully connected layers and non-linear activations (Sigmoid, Tanh, Softmax, ReLU). The library's correctness is rigorously validated by solving the **XOR problem**, a classic benchmark for testing non-linear classification capabilities.
  2. **Unsupervised Learning (Autoencoder):** Leveraging the custom library, we implement an autoencoder architecture designed for image reconstruction on the MNIST dataset. This phase tests the library's ability to handle higher-dimensional data and optimization landscapes more complex than simple logic gates.
  3. **Transfer Learning & Classification:** The final phase explores feature extraction. By utilizing the trained encoder's latent space representations, we

train a Support Vector Machine (SVM) to classify handwritten digits. This demonstrates how neural networks can serve as powerful feature extractors for traditional machine learning algorithms.

- Finally, the project concludes with a critical comparative analysis, benchmarking the custom implementation against industry-standard frameworks to evaluate performance, ease of use, and computational efficiency.

## 1. Library Design and Architecture Choices

- This section details the architectural decisions made during the development of the custom Neural Network library. The library follows a modular Object-Oriented Programming (OOP) design, prioritizing flexibility, readability, and ease of debugging.

### 1.1 Modular Architecture

- The library is structured into distinct modules, each handling a specific aspect of the neural network pipeline. This separation of concerns allows for easy extension (e.g., adding new layer types or optimizers) without modifying the core logic.
  - **Layer Abstraction (Layer.py):** A base abstract class `layer` defines the contract for all network components. It enforces the implementation of two critical methods:
    - `forward(x)`: Computes the output given the input and caches necessary data.
    - `backward(grad_output)`: Computes the gradient with respect to the input and weights.
  - **Dense Layer (Layers.py):** The core fully connected layer implementation. It manages its own weight (`W`) and biases (`b`).
    - *Architecture Choice:* We utilized **Xavier/Glorot uniform initialization** for weights to maintain variance across layers, preventing vanishing or exploding gradients during the initial passes.
  - **Activation Functions (Activation.py):** Activations like `Tanh` and `sigmoid` are implemented as subclasses of `layer`. This treats activations as just another step in the computational graph, simplifying the forward/backward flow.
  - **Network Management (Network.py):** The `Network` class acts as the orchestrator. It maintains a list of layers and manages the training loop.

## 1.2 Forward and Backward Propagation Flow

The data flow follows a strict pipeline managed by the `Network` class:

1. **Forward Pass:** Input data flows sequentially through the list of layers. The output of Layer (N) becomes the input of Layer (N+1).
2. **Loss Calculation:** The final output is compared against the true labels using the `MeanSquaredError` class.
3. **Backward Pass:** The gradient of the loss flows in reverse. Each layer's `backward()` method receives the gradient from the subsequent layer, computes its local gradients (stored in `self.dW`, `self.db`), and passes the gradient w.r.t input to the previous layer.
4. **Parameter Update:** The SGD optimizer iterates through all layers and updates parameters using the stored gradients.

## 2. Results from the XOR Test

- The XOR (Exclusive OR) problem is a classic benchmark for neural networks because it is not linearly separable. A single perceptron cannot solve it; it requires a multi-layer architecture with non-linear activation functions.

### 2.1 Experimental Setup

- To validate the library, we constructed a Multi-Layer Perceptron (MLP) with the following architecture:
  - **Input Layer:** 2 Neurons (representing binary inputs 0/1).
  - **Hidden Layer:** 4 Neurons with **Tanh** activation.
  - **Output Layer:** 1 Neuron with **Sigmoid** activation (outputs probability between 0 and 1).
  - **Optimizer:** Stochastic Gradient Descent (SGD) with Learning Rate:  $\eta = 0.5$ .
  - **Loss Function:** Mean Squared Error (MSE).
  - **Training Duration:** 10,000 Epochs.

## 2.2 Training Results

- The network was trained on the standard XOR truth table:  $X = [[0,0], [0,1], [1,0], [1,1]]$ ,  $Y_{\text{true}} = [[0], [1], [1], [0]]$
- **Convergence:** The model successfully converged. The Mean Squared Error started high (approx. 0.104) due to random initialization and decreased steadily, stabilizing near 0.00103 by epoch 10,000.

## 2.3 Final Predictions

- After training, the model was evaluated on the input set. The raw predicted probabilities demonstrate that the network has successfully learned the XOR logic.

<b>Input (A, B)</b>	<b>True Label</b>	<b>Predicted Probability</b>	<b>Rounded Output</b>
(0, 0)	0	0.0228	0
(0, 1)	1	0.9527	1
(1, 0)	1	0.9505	1
(1, 1)	0	0.0556	0

<b>Input (A, B)</b>	<b>True Label</b>	<b>Predicted Probability</b>	<b>Rounded Output</b>
(0, 0)	0	0.0228	0
(0, 1)	1	0.9527	1
(1, 0)	1	0.9505	1
(1, 1)	0	0.0556	0

## 2.4 Conclusion of Validation

- The results confirm that the library's backpropagation algorithm is correctly implemented. The network successfully learned non-linear decision boundaries, proving the correctness of the Dense layer gradients, Tanh/Sigmoid derivatives, and the SGD update rule.