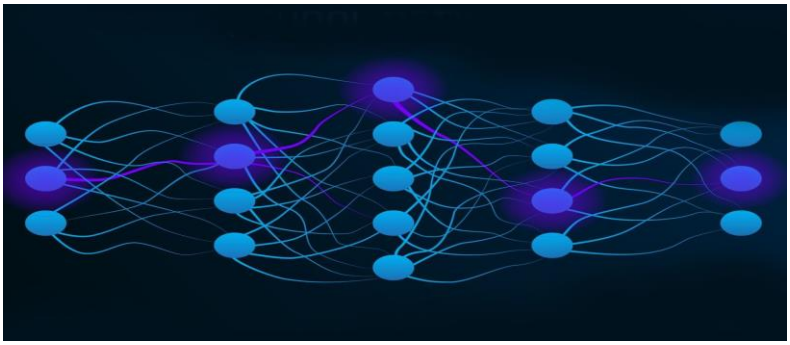
Computational Intelligence.
CSE473s.
Major Task
Part 1 Report.



Team members

Name	ID	section
Nareman Abdelrahman	2101538	1
Noran Tarek Hassan	2101108	1
Haneen Amr Ahmed	2101068	1
Touqa Moustafa Sayed	2100574	1
Ahmed Assem Hassan	2101493	3

Submitted to:

Dr.Hossam eldeen Hassan.

Eng.Abdullah Awadallah

Table of Contents

Introduction

This project focuses on building a complete neural network library from scratch using Python and NumPy. The goal is to understand the core mechanics of neural networks—forward propagation, backward propagation, gradient computation, and optimization—without relying on high-level frameworks like Tensor Flow.

The library includes essential components such as dense layers, activation functions, loss functions, and an SGD optimizer. Its correctness is first validated through the XOR problem, a classic benchmark that requires a non-linear model.

The library is then used to implement and train an autoencoder on the MNIST dataset to learn compact latent representations. These latent features are further used to train an SVM classifier for digit recognition.

Finally, the same architectures are implemented in TensorFlow/Keras to compare performance, training efficiency, and implementation simplicity. This provides a clear contrast between low-level manual implementation and modern deep learning frameworks.

1.What is Neural Network?

Is a computational model in artificial intelligence (AI) and machine learning that is inspired by the structure and function of the human brain. It consists of interconnected nodes (neurons) arranged in layers that process information and learn patterns from data to make predictions or decisions.

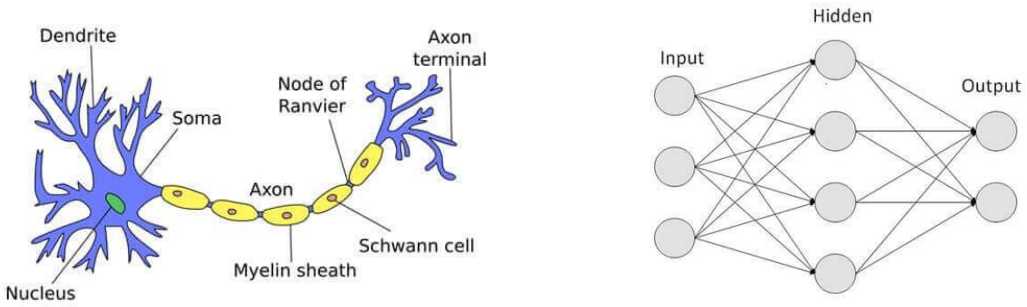


Figure 1 Neural network

Design and Implementation of the Neural Network library:

Moving to the structure and implementation of the custom neural network library developed for this project. The library is fully implemented using Python and NumPy, and follows a modular

design to ensure clarity, reusability, and ease of extension.

---

2. System Structure

The library is organized into separate modules, each responsible for a core component of the neural network framework:

```
lib/
  ├── layers.py
  ├── activations.py
  ├── losses.py
  ├── optimizer.py
  └── network.py
```

- **layers.py** – Implements the base layer structure and the fully connected (Dense) layer.
- **activations.py** – Contains activation functions such as ReLU, Sigmoid, Tanh, and Softmax.
- **losses.py** – Provides the Mean Squared Error (MSE) loss and its gradient.
- **optimizer.py** – Implements the Stochastic Gradient Descent (SGD) optimizer.
- **network.py** – Defines the main Network class managing the forward and backward passes.

This modular architecture ensures that each component is independent, making the library easy to debug, improve, and extend.

2.2 Layer Design

All layers inherit from a common **Layer** base class, which defines the required structure for neural network components.
Each layer implements two key methods:

- **forward**:Computes the layer output given the input.
- **Backward**:Computes the gradient of the loss with respect to the input.

Dense Layer

The fully connected (Dense) layer performs a linear transformation:

$$\mathbf{Y = X * W + b}$$

It stores:

- **W** – weights
- **b** – biases
- **dW**, **db** – gradients computed during backpropagation

This layer forms the core building block for the networks used in the XOR test and MNIST autoencoder.

2.3 Activation Functions

Activation functions introduce non-linearity into the network. Each activation is implemented as a layer with its own forward and backward logic.

Implemented activations include:

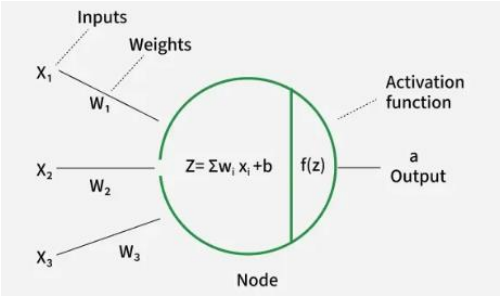


Figure 2 Activation functions

- **ReLU:**

  $$\text{ReLU}(x) = \max(0, x)$$

  Efficient for deep models and frequently used in the encoder.

- **Sigmoid:**

  $$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}$$

  Used in the XOR output layer.

- **Tanh:**
  Outputs values between –1 and 1, useful in the XOR hidden layer.
- **Softmax:**
  Converts logits into probability distributions for multi-class outputs.

Each activation layer stores the input during the forward pass so its derivative can be applied during backpropagation.

Comparison between different activation functions:

Activation	formula	Output range	pros	cons	Use case
Sigmoid	$\frac{1}{1+e^{-x}}$	(0, 1)	Smooth, good for probabilities	Vanishing gradient, not zero-centered	Binary classification output
ReLU	max(0,x)	(-1, 1)	Fast, sparse activation, less vanishing	Can cause dead neurons.	Hidden layers.
Tanh	tanh(x)	[0, ∞)	Zero-centered, stronger gradients	Vanishing gradient at extremes	Hidden layers.
Softmax	$\frac{e^{xi}}{\sum e^{xj}}$	(0,1), sum=1	Outputs class probabilities	Only for multi-class outputs	Output layer for classification

What is forward and backward propagation?

Point of comparison	Forward propagation	Backward propagation
Definition	Input data passes through the network layer by layer to produce an output (prediction).	The network calculates the error, then moves backward to update weights by computing gradients to reduce the error.
Direction	Input → Output	Output → Input
purpose	Compute output/prediction	Update weights to reduce error
Key operation	Weighted sum + activation functions.	Compute gradients via chain rule.
uses	Prediction during training and inference.	Learning/updating model parameters.

2.4 Loss Function

The library uses **Mean Squared Error (MSE)** as the primary loss function:

$$\frac{1}{N}\sum(y_{true} - y_{pred})^2$$

The derivative used for backward propagation is:

$$\frac{d_L}{d_{ypred}} = \frac{2}{N}(y_{true} - y_{pred})$$

MSE is used for both the XOR classification and the autoencoder reconstruction task.

2.5 Optimizer (SGD)

The **Stochastic Gradient Descent (SGD)** optimizer updates model parameters after each backward pass:

$$W = W - \eta\frac{d_L}{d_w}$$

where **η** is the learning rate.

The optimizer processes every trainable layer, applying updates to both weights and biases.

2.6 Network Class

The **Network** class provides a simple, sequential model interface:

- **add(layer)** : adds a new layer to the model.
- **forward(X)** : feeds input through all layers.
- **backward(dout)** : applies backpropagation across all layers in reverse.
- **train (X, y, epochs)** : handles the full training loop.

The class shows the complete forward–backward pipeline and integrates seamlessly with the optimizer.

3.Gradient checking

Gradient checking is used to verify that the backpropagation implementation in the library is correct. Since backpropagation involves many chained derivatives, even small mistakes can lead to incorrect gradients and poor training. Numerical gradient checking provides a reliable way to confirm the correctness of the analytical gradients.

3.1 Numerical Gradient Formula

To validate each parameter W, we approximate its gradient using a small value ε:

```
dL/dW ≈ (L(W + ε) − L(W − ε) ) / (2 * ε)
```

This numerical gradient is then compared to the analytical gradient computed through backpropagation.

3.2 Procedure

1. Perform a forward pass to compute the loss.
2. Compute analytical gradients using backpropagation.
3. Compute numerical gradients by perturbing each parameter.
4. Compare both gradients and measure the difference.

3.3 Results

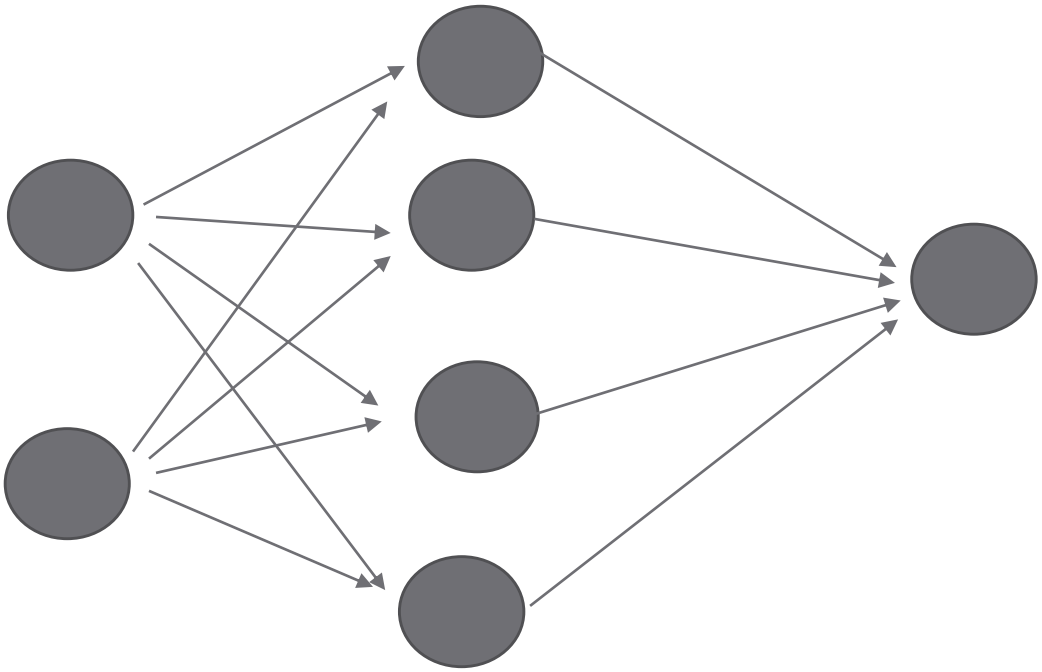
The difference between analytical and numerical gradients was extremely small (typically below $1e^{-6}$), confirming that the backward pass and all layer derivatives were implemented correctly.

4.XOR problem testing:

The XOR problem is a classic benchmark used to verify that a neural network can learn non-linear relationships. Since XOR cannot be solved by a single linear layer, it provides a strong test of whether the implementation of forward propagation, backward propagation, and parameter updates is functioning correctly.

4.1 Problem Definition

- The XOR function has :
- **Input layer:** 2 neurons
- **Hidden layer:** 4 neurons with Tanh activation
- **Output layer:** 1 neuron with Tanh activation



There are two target output approaches for the XOR problem :
First approach:

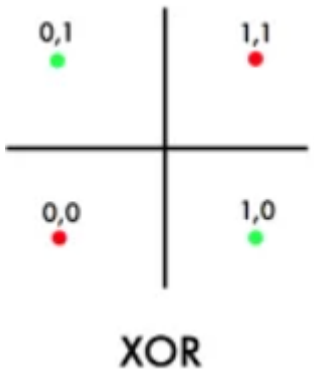
- Output range:[0,1].
- Can suffer from slower convergence due to non-zero-centered output.
- Recommended activation function **Sigmoid** (σ).

Second approach:

- Output range:[-1,1].
- Zero-centered, sometimes faster training
- Recommended activation function **Tanh**

Input	Target
[0,0]	-1
[0,1]	1
[1,0]	1
[1,1]	-1

Input	Target
[0,0]	0
[0,1]	1
[1,0]	1
[1,1]	0



4.2 Network Architecture

A simple multilayer perceptron (MLP) was used:

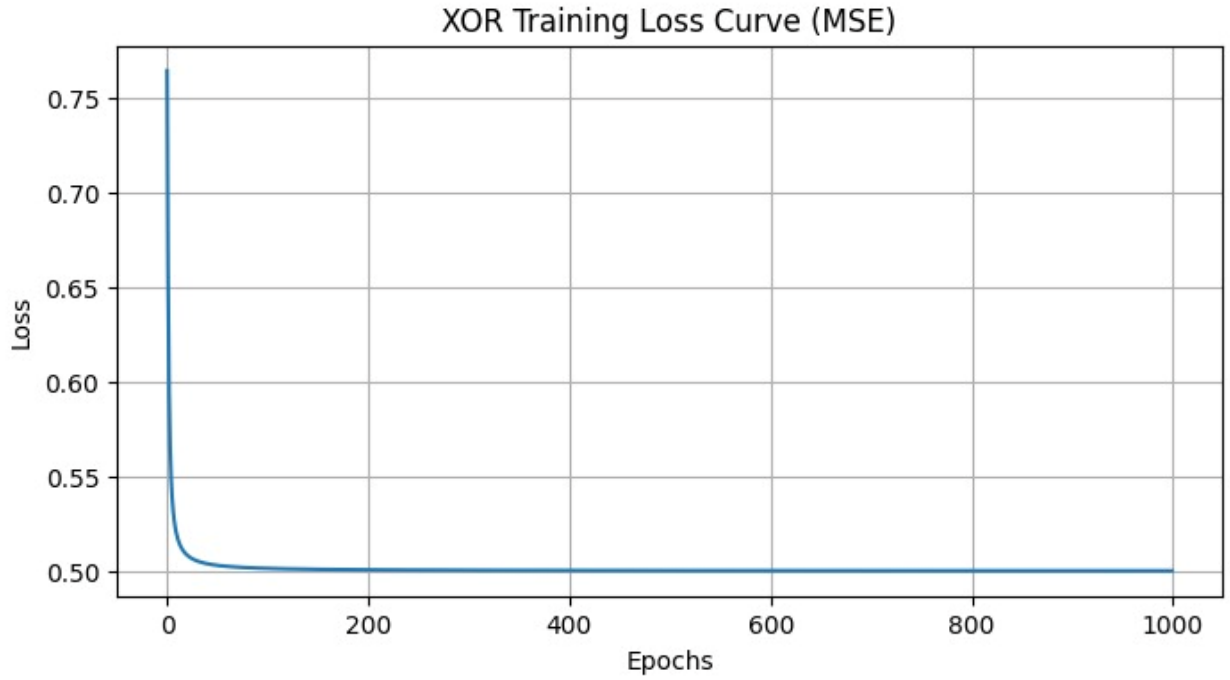
- **Input layer:** 2 neurons
- **Hidden layer:** 4 neurons with Tanh activation
- **Output layer:** 1 neuron with Tanh activation
- **Loss:** Mean Squared Error (MSE)
- **Optimizer:** SGD

This architecture is sufficient for learning the non-linear XOR function.

4.3 Training Setup

- Learning rate: 0.5
- Epochs: 10000

4.4 Results of XOR



Final loss= 0.000171.

5.Tensor flow comparison

TensorFlow/ Keras Implementation for XOR Problem

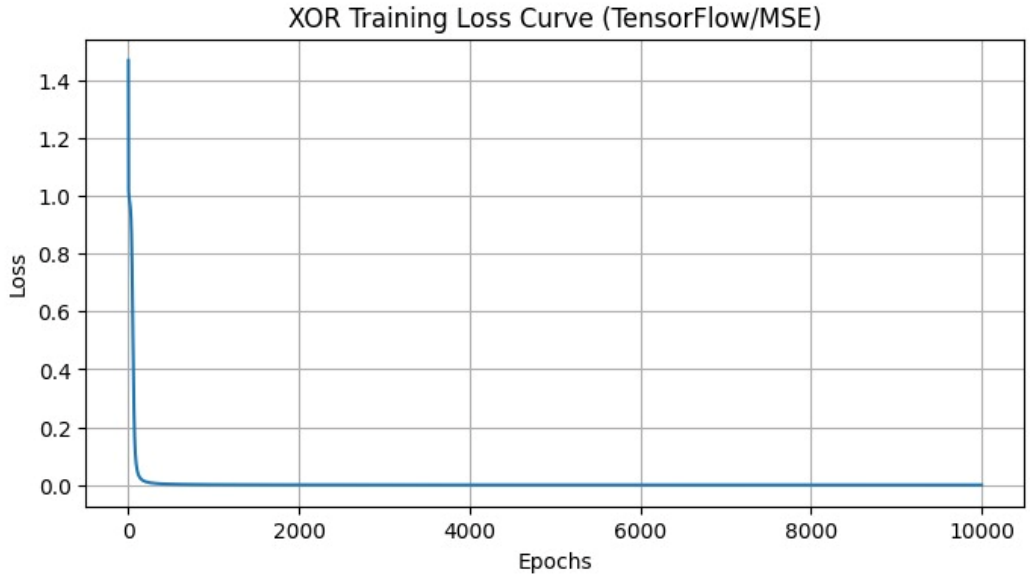
To benchmark our custom neural network library, we implemented the XOR problem using TensorFlow/Keras with the same network architecture (2-4-1) and activation functions (Tanh and Sigmoid).

Implementation:

Using Keras's high-level API, the network was defined with built-in layers and activation functions. This significantly reduced development time compared to manually coding layers, activations, and backpropagation in our library.

Training and Performance:

Training the XOR network in TensorFlow/Keras was faster due to optimized backend computations. The model converged quickly, reaching near-perfect accuracy on all four XOR inputs after a similar number of epochs as our custom implementation.



Final loss= 0.000102.

Comparison:

- **Ease of Use:** Keras greatly simplified model construction and training, requiring less boilerplate code and fewer debugging steps.
- **Speed:** TensorFlow's optimizations led to reduced training time, especially noticeable on larger datasets.
- **Accuracy:** Both implementations produced accurate XOR outputs, validating the correctness of our library's core algorithms.

6.Challenges faced and lessons learned

- **Non-linearity:** XOR is not linearly separable, so the network must have a hidden layer with a non-linear activation (Tanh /ReLU).

- **Backpropagation correctness:** Any mistake in gradients will stop the XOR network from learning.

- **Hyperparameters :** Learning rate, number of epochs, and weight initialization strongly affect convergence.

- **Small network sensitivity:** A 2-4-1 network is simple, so it is very sensitive to bad initialization or poor activation choices.

- **Activation function choice:** Sigmoid may saturate; Tanh usually works better for XOR.

We did many trials trying to decrese the loss by decreasing learning rate and increasing the epochs and here is a results before tunning.