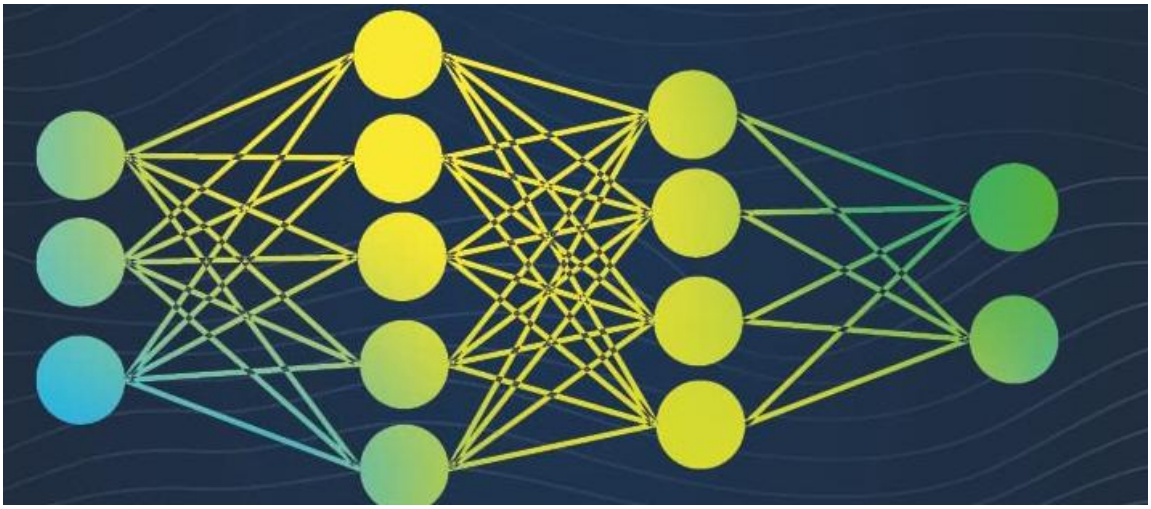
Computational Intelligence Project

Part one



Names	ID	Sec
Abdelrahman Essam Niazy	2101502	2
Ahmed Ashraf Salamah	2000229	2
Nabil Ali Ibrahim	2101592	2
Mohamed Mahmoud Aljamal	2002612	2
Youssef Ehab Mohamed	2101375	2

Delivered to: Dr. Hossam El-Din Hassan

Eng. Abdallah Mohamed Mahmoud

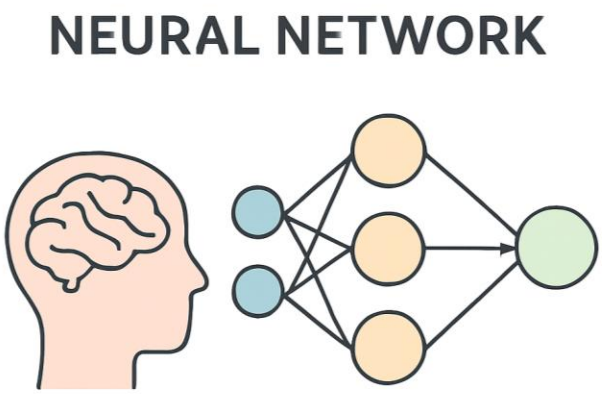
Eng. Dina Zakaria Mahmaud

Contents

Figures

Project Overview

This project focuses on building a foundational neural network library completely from scratch using only Python and NumPy. The goal is to gain a deeper understanding of how neural networks operate at the mathematical and algorithmic levels.



In **Part 1**, the objective was to implement the core components that form the basis of any neural network system. This included creating modular classes for layers, activation functions, loss functions, and an optimizer, as well as designing a simple Sequential model to manage forward and backward propagation. Once the library structure was completed, it was validated through the classic XOR classification problem—a well-known test that cannot be solved by a single linear model and therefore requires a multi-layer neural network.

To verify correctness, the implemented network was trained on the four XOR input-output pairs using a small multilayer perceptron (MLP) with nonlinear activations and gradient-based optimization. Successful convergence and correct predictions demonstrated that the forward pass, backpropagation algorithm, numerical computations, and parameter updates were functioning properly.

Overall, Part 1 established a working neural network library with essential functionality, setting the foundation for the more advanced tasks in the next phases of the project, including building an autoencoder and performing latent-space classification.

Library Architecture

Main Components

- ❖ Layer
  - forward(x)
  - ° backward(grad)
  - ° Holds its own parameters and gradients
- ❖ Model
  - ° Maintains an ordered list of layers
  - Handles the full forward/backward pass
  - Manages the training loop
- ❖ Loss
  - ° Computes loss value
  - Returns gradient wrt predictions
- ❖ Optimizer
  - Updates parameters of all layers

This structure allowed plugging in the autoencoder, XOR network, and MLP with Built-in code.

Figure 1Input Layer

W, b

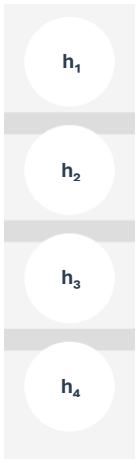


Figure 2Hidden Layer

W, b

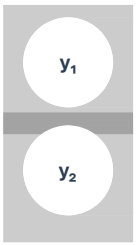


Figure 3Output Layer

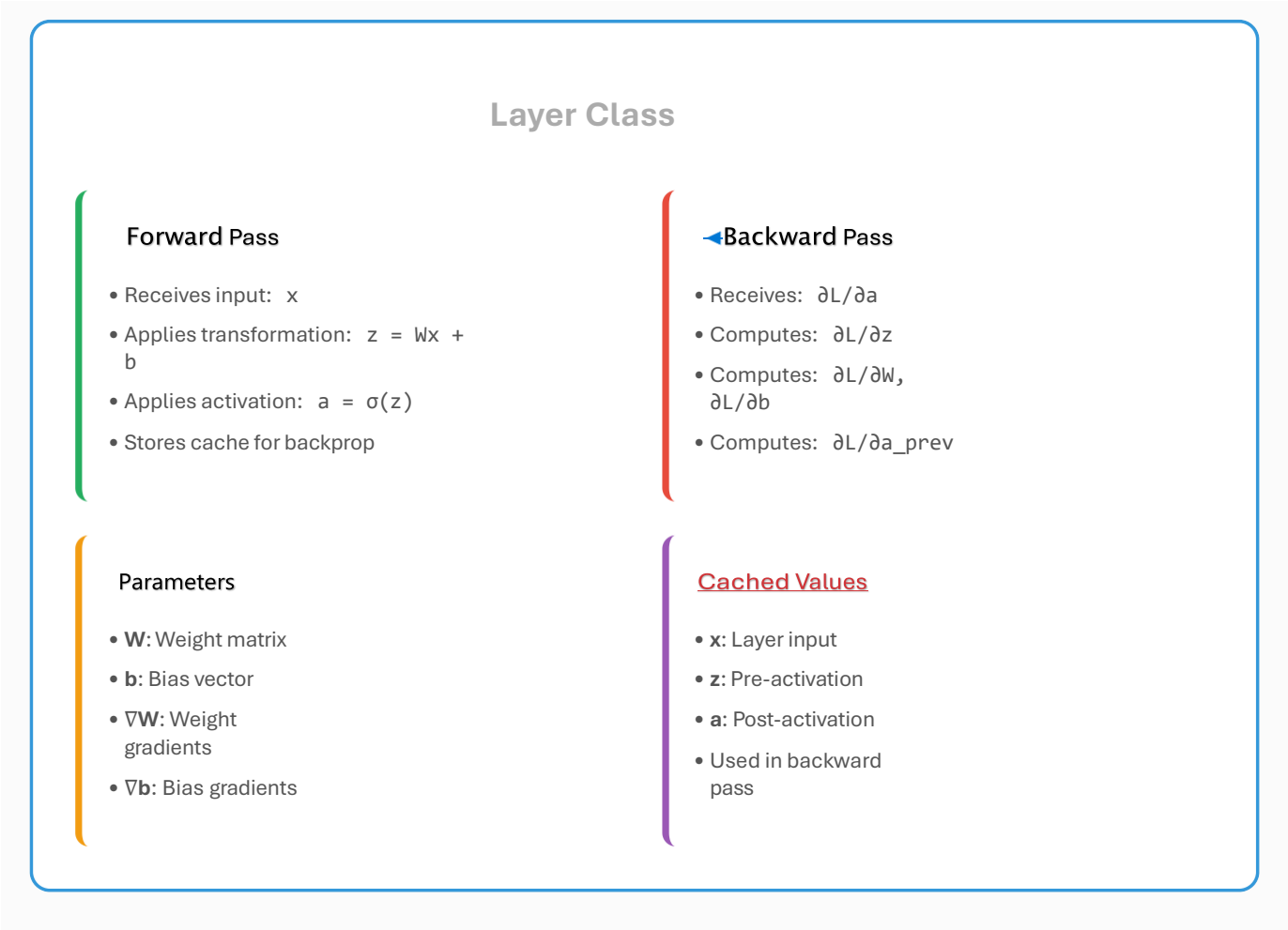


Figure 4 Layer Architecture

Figure 5 visualization of layers

Training Process:

1. Loss computes gradient
2. Gradient flows through backprop in model
3. Optimizer updates all parameters

Mathematical backprop design

Backpropagation in our neural network library follows the chain rule applied layer-by-layer.
Each layer acts as a differentiable function that receives an input, produces an output during the forward pass, and computes local gradients during the backward pass using quantities stored from the forward computation.

We first present the derivation for the fully connected (Dense) layer, then connect it to the activation functions, the loss function, and the global chain rule that propagates gradients through the entire network.

1. Fully Connected (Dense) Layer

For a Dense layer, the forward transformation is:

$$z = XW + b$$
$$a = \sigma(z)$$

Where:

- $X \in \mathbb{R}^{N \times d_{in}}$: input minibatch
- $W \in \mathbb{R}^{d_{in} \times d_{out}}$: weights
- $b \in \mathbb{R}^{d_{out}}$: bias
- $z \in \mathbb{R}^{N \times d_{out}}$: pre-activation
- $a \in \mathbb{R}^{N \times d_{out}}$: activation output
- N: batch size

During backprop, we assume that we already know:

$$\frac{\partial L}{\partial a}$$

and our goal is to compute:

- $\frac{\partial L}{\partial z}$
- $\frac{\partial L}{\partial W}$
- $\frac{\partial L}{\partial b}$
- $\frac{\partial L}{\partial X}$

1.1 Derivative Through the Activation Function

Using the chain rule:

$$\delta = \frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \odot \sigma'(z)$$

Where:

- $\odot$is elementwise multiplication
- $\delta$is the standard backprop "error signal" for this layer

This step applies per-element derivatives of the chosen activation function.

2. Gradients for Weights and Bias

2.1 Gradient w.r.t. weights

Starting from:

$$z = XW + b$$

The derivative of $z_{i,j}$with respect to $W_{k,j}$is:

$$\frac{\partial z_{i,j}}{\partial W_{k,j}} = X_{i,k}$$

Thus:

$$\frac{\partial L}{\partial W_{k,j}} = \sum_{i=1}^{N} X_{i,k}\, \delta_{i,j}$$

Vectorizing this:

$$\frac{\partial L}{\partial W} = X^{T}\delta$$

This is the weight gradient used in the SGD optimizer.

2.2 Gradient w.r.t. bias

Because:

$$z_{i,j} = \sum_{k} X_{i,k} W_{k,j} + b_j$$

We have:

$$\frac{\partial z_{i,j}}{\partial b_j} = 1$$

So:

$$\frac{\partial L}{\partial b_j} = \sum_{i=1}^{N} \delta_{i,j}$$

Vectorized:

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{N} \delta_i$$

3. Gradient Passed to Previous Layer

The gradient needed by the previous layer is:

$$\frac{\partial L}{\partial X} = \delta W^T$$

This completes the Dense layer's backward computation.

Thus, each Dense layer returns:

- $\delta_x = \frac{\partial L}{\partial X}$
- $\frac{\partial L}{\partial W}$
- $\frac{\partial L}{\partial b}$

These are stored for use in the optimizer.

4. Activation Function Derivatives

The activation derivatives used in the chain rule are:

Sigmoid

Tanh

ReLU

Each activation layer applies elementwise derivatives:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{z}} = \frac{\partial \mathrm{L}}{\partial \mathrm{a}} \odot \mathrm{f}'(\mathrm{z})$$

5. Loss Function Gradient (MSE)

The Mean Squared Error loss is:

$$L = \frac{1}{N} \sum_{i=1}^{N} \quad \| \, y_i - \hat{y}_i \, \|^2$$

Derivative with respect to the model output:

$$\frac{\partial L}{\partial \hat{y}} = \frac{2}{N} (\hat{y} - y)$$

This gradient is the **starting point** for backpropagation through the entire network.

6. Layer-to-Layer Gradient Flow (Chain Rule)

Backprop flows from the last layer to the first as:

$$\frac{\partial L}{\partial a^{(l)}} = \frac{\partial L}{\partial z^{(l)}} \odot \sigma'(z^{(l)})$$
$$\frac{\partial L}{\partial z^{(l)}} = \frac{\partial L}{\partial X^{(l+1)}}$$

And:

$$\frac{\partial L}{\partial X^{(l)}} = \delta^{(l)} W^{(l)T}$$

This systematic chain rule ensures compatibility with the Sequential model design.

7. Gradient Checking (Verification of Backprop)

To ensure correctness, we compare analytical gradients to numerical gradients:

$$g_{\text{num}} = \frac{L(\theta + \epsilon) - L(\theta - \epsilon)}{2\epsilon}$$

Analytical gradient:

$$g_{\text{analytic}} = \frac{\partial L}{\partial \theta}$$

Relative error:

$$\text{error} = \frac{\| g_{\text{analytic}} - g_{\text{num}} \|}{\| g_{\text{analytic}} \| + \| g_{\text{num}} \| + 10^{-8}}$$

We verified that the relative error for Dense layer parameters is in the order of:

$$10^{-11}$$

confirming that our backpropagation implementation is mathematically correct.

Milestone 1 results

Gradient Checking test:

Test Procedure:

This Test was carried out to ensure the accuracy of the gradients calculated by the library. it was done by first calculating the gradients via automatic differentiation using Equation 1

$$\frac{\partial f}{\partial x}|_{x=x_0} = \frac{f(x+\epsilon) - f(x-\epsilon)}{2 \times \epsilon}$$

Equation 1 (Automatic Differentiation)

This gradient was then compared with the gradient calculated by the library.

Test Results

The test was carried out on a 3-layer network shown in Figure 6

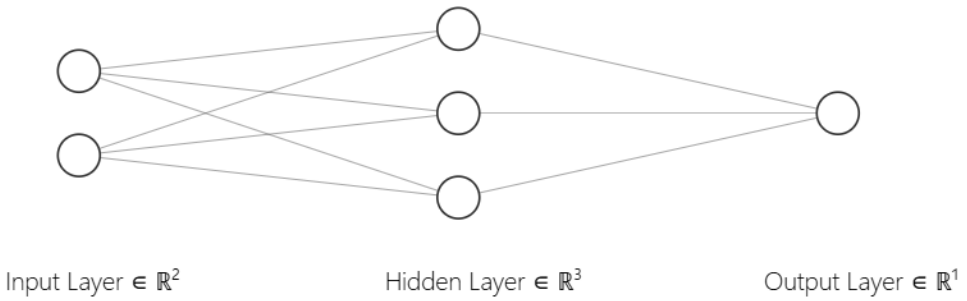


Figure 6 Neural Network used in gradient test

The difference between the norm of the gradients calculated by the library and the gradients calculated numerically are tabulated below and the gradients have been vectorized, so that the hidden and output layers each have 2 parameter gradients one with respect to the weights while the other is with respect to the biases

Gradient with respect to	Error
Hidden layer's weights	5.722442×10^{-11}
Hidden layer's biases	4.967208×10^{-11}
Output layer's weights	1.571804×10^{-11}
Output layer's biases	6.136082×10^{-12}

XOR test Result

Test Procedure:

A neural network was created to solve the XOR problem with the following truth table

X_1	X_2	$Y_{desired}$
0	0	0
0	1	1
1	0	1
1	1	0

Test Results:

The neural network created had the architecture shown in

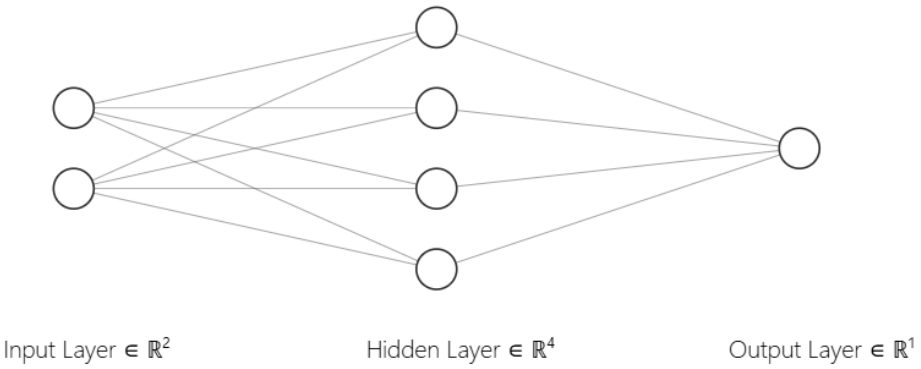


Figure 7 Neural Network used for XOR problem

And it was able to achieve the exact desired output as shown below

X_1	X_2	$Y_{Predicted}$
0	0	0
0	1	1
1	0	1
1	1	0

Challenges + next milestone plan

5.1 Challenges Faced

The journey of building a library from scratch was not without its difficulties. The most significant challenges included:

• **Debugging Backpropagation:** This was the most complex and error-prone phase. Ensuring the correctness of gradient calculations and their flow through multiple layers required meticulous attention to detail, extensive testing, and a deep understanding of the chain rule.

• **Numerical Stability:** During training, we encountered issues with vanishing and exploding gradients. This required careful initialization of weights and tuning of the learning rate to ensure stable convergence.

• **Hyperparameter Tuning:** Without automated tools, finding the optimal set of hyperparameters (learning rate, batch size, layer dimensions, number of epochs) was a manual and iterative process that highlighted the sensitivity of neural networks to these settings.

5.2 Milestone 2 Plan

- In Milestone 2, we will extend our neural network library to train a full **autoencoder** on the MNIST dataset.
- The encoder will compress each 28×28 image into a low-dimensional latent vector, and the decoder will reconstruct the image from this representation.
- After training, we will extract latent vectors for all MNIST samples and evaluate how well this learned representation separates digit classes by training an external **SVM classifier** on the latent space.
- We will report reconstruction quality, training loss curves, and SVM accuracy.
- Finally, we will implement the same autoencoder using **TensorFlow/Keras** to provide a performance and quality comparison between our custom NumPy-based library and a modern deep learning framework. This milestone validates that our library can train deeper models, learn meaningful representations, and generalize to real datasets.