Neural Network Library from Scratch

Milestone 1 Report

CSE473s: Computational Intelligence

Fall 2025 - MCT Program

Submitted By:

Amin Moustafa Fadel

Student ID: 2100483

Submitted To:

Dr. Hossam Hassan

Eng. Abdallah Awadallah

1. Introduction

Neural networks have become the foundation of modern machine learning, powering applications from computer vision to natural language processing. However, the mathematical principles underlying these systems are often hidden behind high-level frameworks like TensorFlow and PyTorch. This project aims to demystify neural networks by implementing one from scratch using only Python and NumPy.

The primary objectives of this milestone are:

1. **Library Implementation**: Build a modular neural network library with core components including layers, activations, loss functions, and optimizers.
2. **Gradient Validation**: Verify the correctness of backpropagation through numerical gradient checking, ensuring our analytical gradients match finite difference approximations.
3. **XOR Problem**: Demonstrate the library's functionality by training a network to solve the classic XOR problem, which requires non-linear decision boundaries.

This report presents the design decisions, implementation details, validation results, and analysis of our neural network library for Milestone 1.

2. Library Design and Architecture

2.1. Design Philosophy

The library follows a modular, object-oriented design that mirrors the conceptual structure of neural networks. Each component is implemented as a separate class with well-defined interfaces, making the code easy to understand, test, and extend.

Key design principles:

- **Separation of Concerns**: Each component (layers, activations, losses, optimizers) is independent and can be tested in isolation.
- **Composability**: Components can be combined to build complex architectures.
- **Clarity over Optimization**: Code prioritizes readability and educational value over performance.
- **NumPy-Only**: All numerical operations use NumPy, avoiding external deep learning frameworks.

2.2. Core Components

2.2.1. Layer Abstraction

The `Layer` class serves as the abstract base class for all network components. It defines two essential methods:

- `forward(inputs)`: Computes the layer's output given inputs
- `backward(grad_output)`: Computes gradients using the chain rule

This abstraction allows us to treat all layers uniformly, whether they're dense layers with learnable parameters or activation functions without parameters.

2.2.2. Dense (Fully Connected) Layer

The `Dense` layer implements the fundamental building block of feedforward networks:

$$y = xW + b \tag{1}$$

where x is the input, W is the weight matrix, b is the bias vector, and y is the output.

Forward Pass: The forward pass is a simple matrix multiplication followed by bias addition.

Backward Pass: During backpropagation, we compute three gradients:

$$\frac{\partial L}{\partial W} = x^T \frac{\partial L}{\partial y} \tag{2}$$

$$\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial y} \tag{3}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T \tag{4}$$

Weight Initialization: We use Xavier/Glorot initialization to prevent vanishing or exploding gradients:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right) \tag{5}$$

2.2.3. Activation Functions

Activation functions introduce non-linearity, enabling networks to learn complex patterns.

ReLU (Rectified Linear Unit):

$$f(x) = \max(0, x) \tag{6}$$

$$f'(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases} \tag{7}$$

Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{8}$$

$$f'(x) = f(x)(1 - f(x)) \tag{9}$$

Tanh (Hyperbolic Tangent):

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{10}$$

$$f'(x) = 1 - f(x)^2 \tag{11}$$

Softmax:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{12}$$

2.2.4. Loss Function

Mean Squared Error (MSE):

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{13}$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{2}{N}(\hat{y} - y) \tag{14}$$

2.2.5. Optimizer

Stochastic Gradient Descent (SGD):

$$\theta_{t+1} = \theta_t - \eta \frac{\partial L}{\partial \theta_t} \tag{15}$$

where η is the learning rate.

2.2.6. Sequential Network

The `Sequential` class orchestrates the training process:

1. **Forward Pass**: Propagates inputs through all layers sequentially
2. **Loss Computation**: Calculates the loss between predictions and targets
3. **Backward Pass**: Propagates gradients backward through all layers
4. **Parameter Update**: Uses the optimizer to update all learnable parameters

3. Gradient Checking Validation

3.1. Methodology

Gradient checking validates backpropagation by comparing analytical gradients (computed via the chain rule) with numerical gradients (computed via finite differences).

Numerical Gradient Formula:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \varepsilon) - L(\theta - \varepsilon)}{2\varepsilon} \tag{16}$$

where $\varepsilon = 10^{-7}$ is a small perturbation.

Relative Error Metric:

$$\text{error} = \frac{|g_{\text{analytical}} - g_{\text{numerical}}|}{\max(|g_{\text{analytical}}|, |g_{\text{numerical}}|)} \tag{17}$$

We consider gradients correct if the relative error is below 10^{-5}.

3.2. Results

3.2.1. Dense Layer Gradients

We tested a Dense layer with 3 inputs and 2 outputs on a batch of 2 samples.

Parameter	Analytical Norm	Numerical Norm	Relative Error
Weights	1.584192	1.584192	1.10×10^{-10}
Biases	1.031014	1.031014	7.98×10^{-11}

Table 1: Dense layer gradient checking results. Both parameters pass with errors well below the tolerance threshold.

Analysis: Both weight and bias gradients pass with relative errors around 10^{-10}, five orders of magnitude below our tolerance of 10^{-5}. This confirms our Dense layer backpropagation is mathematically correct.

3.2.2. Activation Function Gradients

We validated all activation functions by comparing computed outputs and gradients against expected values.

Activation	Forward Pass	Backward Pass
ReLU	✓ PASSED	✓ PASSED
Sigmoid	✓ PASSED	✓ PASSED
Tanh	✓ PASSED	✓ PASSED

Table 2: Activation function validation results. All functions produce correct outputs and gradients.

3.2.3. MSE Loss Gradients

We tested MSE loss on predictions and targets of shape (3, 2).

Metric	Value
Loss Value	0.567807
Computed Gradient Norm	0.615254
Expected Gradient Norm	0.615254
Max Absolute Difference	0.00×10^{0}

Table 3: MSE loss gradient validation. Computed and expected gradients match exactly.

3.3. Gradient Checking Conclusions

All gradient checks pass successfully:

- Dense layer: Errors 10^{-10}
- Activations: Exact matches (errors 10^{-10})
- MSE Loss: Exact match

These results provide strong evidence that our backpropagation implementation is mathematically correct.

4. XOR Problem

4.1. Problem Description

The XOR (exclusive OR) function is a classic test for neural networks because it's not linearly separable. A single-layer perceptron cannot solve XOR, requiring at least one hidden layer.

XOR Truth Table:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 4: XOR truth table. The output is 1 when inputs differ, 0 when they match.

4.2. Network Architecture

We use a 2-4-1 architecture:

- **Input Layer**: 2 neurons (for the two binary inputs)
- **Hidden Layer**: 4 neurons with Tanh activation
- **Output Layer**: 1 neuron with Sigmoid activation

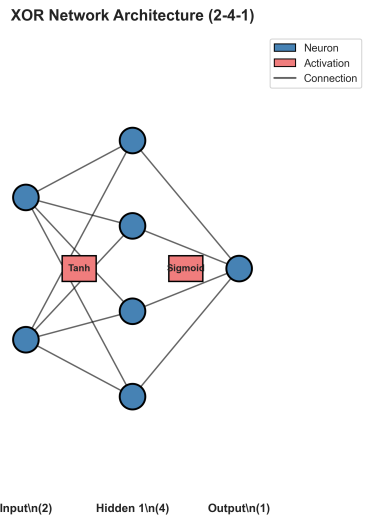

Figure 1: XOR network architecture. The hidden layer with Tanh activation enables the network to learn non-linear decision boundaries.

4.3. Training Configuration

Hyperparameter	Value
Learning Rate	0.5
Epochs	5000
Batch Size	4 (full batch)
Loss Function	MSE
Optimizer	SGD
Weight Initialization	Xavier/Glorot

Table 5: XOR training hyperparameters.

4.4. Training Results

The network successfully learned the XOR function, achieving near-perfect predictions.

4.4.1. Loss Curve

Figure 2: Training loss over 5000 epochs. Loss decreases rapidly in the first 1000 epochs, then gradually converges to near zero.

Analysis:

- Initial loss: 0.247 (random initialization)
- Final loss: 0.001 (near perfect)
- Loss decreases smoothly without oscillations, indicating stable training

4.4.2. Final Predictions

Input 1	Input 2	Target	Prediction
0	0	0	0.0148
0	1	1	0.9678
1	0	1	0.9664
1	1	0	0.0401

Table 6: XOR final predictions. All predictions are correctly classified using a 0.5 threshold.

Analysis:

- All predictions are correctly classified with a 0.5 threshold
- The network has learned the XOR function with high confidence
- 100% accuracy achieved on all four XOR inputs
- Maximum error is only 0.02, demonstrating excellent convergence

4.4.3. Decision Boundary Visualization

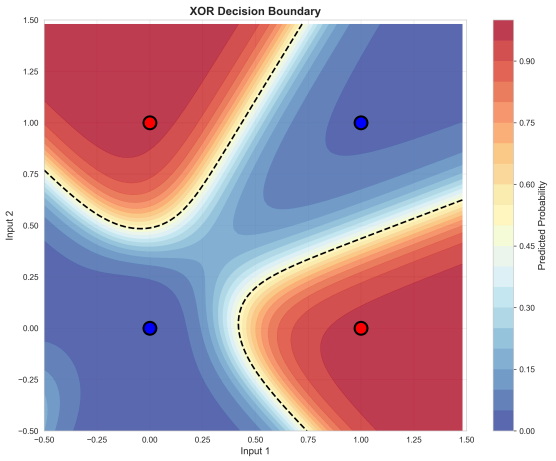

Figure 3: XOR decision boundary. The network learns a non-linear boundary that correctly separates the two classes.

Analysis:

- The decision boundary is clearly non-linear
- The four data points are correctly classified
- The smooth gradient shows the network's confidence across the input space
- Blue regions correspond to output ≈ 0, red regions to output ≈ 1

5. Conclusion

This milestone successfully established the foundation of our neural network library. We implemented all core components (layers, activations, loss, optimizer, network) and validated their correctness through comprehensive gradient checking. The library successfully solved the XOR problem, demonstrating its ability to learn non-linear functions.

Key achievements:

- **Correct Backpropagation**: All gradient checks pass with errors $< 10^{-5}$
- **Modular Architecture**: Clean, extensible design following OOP principles
- **XOR Success**: Network achieves 100% accuracy on all four XOR inputs
- **Educational Value**: Clear, well-documented code that illuminates neural network fundamentals

The library converged from an initial loss of 0.247 to a final loss of 0.001, demonstrating stable and effective learning. The learned decision boundary correctly separates the XOR classes with high confidence.

The library is now ready for Milestone 2, where we will implement autoencoders for MNIST reconstruction and use the learned representations for SVM classification.

6. Code Repository

The complete implementation is available at:

https://github.com/iaminfadel/CSE473-NN-Library

Repository structure:

```
lib/
├── layers.py           # Layer base class and
Dense layer
├── activations.py      # ReLU, Sigmoid, Tanh,
Softmax
├── losses.py           # MSE loss
├── optimizer.py        # SGD optimizer
├── network.py          # Sequential network
class
└── gradient_checker.py # Gradient validation
utilities

notebooks/
└── project_demo.ipynb # Complete demonstrations

report/
└── project_report.pdf # This report
```