
**AIN SHAMS UNIVERSITY**
**FACULTY OF ENGINEERING**
MECHATRONICS ENGINEERING DEPARTMENT
CSE473s: Computational Intelligence
Fall 2025

# Project Report

## Library Implementation, Autoencoder and Latent Space Classification

### Team 15



| Name | ID | Section |
|---|---|---|
| Abdelrhman Tarek Mohamed | 2101547 | Sec.1 |
| Ahmed Mahmoud AbdelAzeem | 2100718 | Sec.1 |
| El mahdy Salih | 2101875 | Sec.1 |
| Mahmoud Atef | 2001313 | Sec.1 |
| Zaid Reda Farouk | 2101603 | Sec.1 |



**Submitted to:**

Eng. Abdallah Mohamed

# Contents

**1 Introduction** **2**

**2 Library Architecture and Design** **2**
2.1 Repository Structure . . . 2
2.2 Layer Abstraction (`lib.layers`) . . . 2
2.3 Activation Functions (`lib.activations`) . . . 3
2.4 Loss Function (`lib.losses`) . . . 3
2.5 Optimizer (`lib.optimizer`) . . . 3
2.6 Network Model (`lib.network`) . . . 3

**3 Validation and Testing** **3**
3.1 Gradient Checking . . . 3
3.2 XOR Problem Validation . . . 4
3.2.1 Network Architecture . . . 4
3.2.2 Training Parameters . . . 4
3.2.3 Results . . . 4

**4 Part 2: Advanced Applications** **5**
4.1 Autoencoder Implementation . . . 5
4.1.1 Architecture . . . 5
4.1.2 Training Configuration . . . 5
4.1.3 Results . . . 5
4.2 Latent Space Classification (SVM) . . . 5
4.2.1 Methodology . . . 5
4.2.2 Classification Results . . . 6
4.3 Comparative Analysis (vs. TensorFlow/Keras) . . . 6

**5 Conclusion** **6**

# 1 Introduction

The objective of this project is to build a foundational neural network library from scratch and demonstrate its capabilities on advanced tasks. "Part 1" involves implementing core components (layers, activations, optimizer) and validating them with gradient checking and the XOR problem. "Part 2" extends this by implementing an Autoencoder for the MNIST dataset to learn latent representations, which are then used for classification via Support Vector Machines (SVM). Finally, a benchmark comparison against TensorFlow/Keras is performed to evaluate the library's efficiency.

# 2 Library Architecture and Design

The library is structured into modular Python files within the `lib/` directory. The design favors simplicity and clarity while ensuring mathematical correctness.

## 2.1 Repository Structure

The following directory tree illustrates the organization of the project:

```
.
        .gitignore
        README.md
        requirements.txt
        lib/
              __init__.py
              layers.py        # Dense layer implementation
              activations.py   # ReLU, Sigmoid, Tanh
              losses.py        # MSE Loss
              optimizer.py     # SGD Optimizer
              network.py       # Sequential model class
        notebooks/
              project_demo.ipynb  # Demos: Gradient Check, XOR,
  Autoencoder
        report/
            Part1_Report.pdf
```

## 2.2 Layer Abstraction (`lib.layers`)

All layers inherit from a base `Layer` class, which defines the interface for `forward` and `backward` passes.

- **Dense Layer**: Implements a fully connected layer.
- **Initialization**: He initialization is used for weights to verify better convergence ($W \sim \mathcal{N}(0, \sqrt{2/n_{in}})$).
- **Gradient Accumulation**: The layer supports gradient accumulation to handle batch processing correctly.

### 2.3 Activation Functions (`lib.activations`)

Activation functions are implemented as subclasses of `Layer` to integrate seamlessly into the computation graph.

- **ReLU**: $f(x) = \max(0, x)$
- **Sigmoid**: $f(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $f(x) = \tanh(x)$

### 2.4 Loss Function (`lib.losses`)

The **Mean Squared Error (MSE)** is implemented to measure the discrepancy between predictions and targets.

$$L = \frac{1}{N}\sum(Y_{true} - Y_{pred})^2$$

The class provides both `loss` (forward) and `loss_prime` (gradient w.r.t prediction) methods.

### 2.5 Optimizer (`lib.optimizer`)

**Stochastic Gradient Descent (SGD)** is used to update parameters. The `step` method updates weights and biases using the computed gradients and a specified learning rate.

$$W_{new} = W_{old} - \eta\frac{\partial L}{\partial W}$$

### 2.6 Network Model (`lib.network`)

The `Sequential` class acts as a container for layers. It manages:

- **Forward Pass**: Sequentially passing input through all layers.
- **Training Loop**: Iterating through epochs, performing forward/backward passes, and invoking the optimizer.

## 3 Validation and Testing

### 3.1 Gradient Checking

To ensure the correctness of the analytical backpropagation, we performed a gradient check comparing the analytical gradients against numerical gradients computed via finite differences.

$$\frac{\partial L}{\partial W} \approx \frac{L(W+\epsilon) - L(W-\epsilon)}{2\epsilon}$$

**Results:**

- **Analytic Gradient**: `[[-2.07469049], [-0.43725958]]`
- **Relative Difference**: $2.81 \times 10^{-12}$

The extremely small difference confirms the implementation of backpropagation is correct.

## 3.2 XOR Problem Validation

The XOR function is a classic non-linear classification problem that requires a multi-layer perceptron to solve.

### 3.2.1 Network Architecture

A 2-layer network was constructed:

1. **Input Layer**: 2 neurons
2. **Hidden Layer**: Dense(16 units) + Tanh activation
3. **Output Layer**: Dense(1 unit) + Sigmoid activation

### 3.2.2 Training Parameters

- **Epochs**: 10,000
- **Learning Rate**: 1.0
- **Loss Function**: MSE

### 3.2.3 Results

The network successfully converged, achieving a final loss of approximately **0.00003**.

**Training Log** Below is a snippet of the training progress showing the reduction in error over epochs:

```
Training XOR Network
==================================================
Epoch 1/10000       error=0.270882
...
Epoch 1001/10000    error=0.000429
...
Epoch 5001/10000    error=0.000064
...
Epoch 9901/10000    error=0.000030
==================================================
Training Complete
```

Table 1: Final Predictions on XOR Dataset

| Input (x1, x2) | Truth (y) | Predicted Probability | Rounded Prediction |
|---|---|---|---|
| (0, 0) | 0 | 0.003440 | 0 |
| (0, 1) | 1 | 0.994145 | 1 |
| (1, 0) | 1 | 0.994501 | 1 |
| (1, 1) | 0 | 0.006622 | 0 |

The model achieved **100% accuracy** on the XOR dataset, validating the capability of the library to learn non-linear decision boundaries.

# 4 Part 2: Advanced Applications

In this phase, we utilized the custom library to solve a dimensionality reduction problem using an Autoencoder and performed classification on the learned latent space.

## 4.1 Autoencoder Implementation

We implemented an undercomplete autoencoder to compress MNIST images (28x28 pixels) into a lower-dimensional latent space.

### 4.1.1 Architecture

The network compresses the 784-dimensional input into a 128-dimensional latent vector and then reconstructs it.

- **Encoder**: Input(784) $\rightarrow$ Dense(128) $\rightarrow$ ReLU
- **Decoder**: Dense(128) $\rightarrow$ Dense(784) $\rightarrow$ Sigmoid

### 4.1.2 Training Configuration

- **Dataset**: 10,000 samples from MNIST (normalized to [0, 1])
- **Optimizer**: SGD (Learning Rate = 0.5)
- **Batch Size**: 64
- **Epochs**: 100

### 4.1.3 Results

The autoencoder successfully learned to reconstruct digits, achieving a final Mean Squared Error (MSE) of **0.00468**. The reconstruction quality was verified visually, with digit shapes being clearly preserved.

## 4.2 Latent Space Classification (SVM)

To validate the quality of the learned representations, we extracted the 128-dimensional latent vectors from the trained encoder and used them to train a Support Vector Machine (SVM) classifier.

### 4.2.1 Methodology

1. **Feature Extraction**: Pass 60,000 training images through the Encoder to get 128-dim vectors.
2. **Classifier**: Train an sklearn `SVC` (RBF kernel) on these vectors.
3. **Evaluation**: Test on 10,000 test images.

### 4.2.2 Classification Results

The SVM classifier achieved an accuracy of **96.76%** on the test set. This high accuracy demonstrates that the custom autoencoder successfully captured the semantic features of the handwritten digits in the compressed latent space.

## 4.3 Comparative Analysis (vs. TensorFlow/Keras)

We benchmarked our custom library against the industry-standard TensorFlow/Keras framework to evaluate performance and implementation effort.

Table 2: Benchmark: Custom Library vs. TensorFlow/Keras

| Metric | My Library (Custom) | TensorFlow / Keras |
|---|---|---|
| **XOR Final Loss** | 0.00043 | 0.00004 |
| **XOR Training Time** | 8.34s | 25.95s |
| **AE Final Loss** | 0.00468 | 0.00238 |
| **AE Reconstruction MSE** | 0.00496 | 0.00200 |
| **AE Training Time** | 960s | 130s |
| **Implementation (LOC)** | ∼100 lines | ∼10 lines |

While Keras is significantly faster for large-scale training (due to optimizations and C++ backend), our custom library produced competitive accuracy and loss convergence, validating the correctness of our mathematical implementation.

# 5 Conclusion

This project successfully demonstrated the implementation of a neural network library from scratch. We verified its core functionality through gradient checking and the XOR problem (Part 1) and extended it to complex unsupervised learning tasks with Autoencoders (Part 2). The ability to achieve **96.76%** accuracy on MNIST using latent features confirms the library's effectiveness. The comparative analysis highlights the trade-offs between educational implementations and production-grade frameworks.