A Neural Network Library and Advanced Applications

Computational Intelligence (CSE473s) – Fall 2025

Team 5
Department of Mechatronics Engineering
Faculty of Engineering
Ain Shams University

Name	**Student ID**
Abdelghaffar Essam	2100428
Ingy Ahmed	2100648
Kareem Saleh	2100909
Mariam Elsebaey	2100260

Submitted to:
Prof. Hossam Hassan
Eng. Abdallah Awdallah

GitHub Repository:
Neural Network Library

December 16, 2025

Abstract

This project presents the implementation of a neural network library from scratch using Python and NumPy. The library is validated through gradient checking, solving the XOR problem, training an autoencoder on the MNIST dataset, and performing digit classification using a Support Vector Machine (SVM) on learned latent representations. Results are compared against TensorFlow/Keras implementations to evaluate performance and correctness.

Contents

1 Introduction **4**

2 Library Design and Architecture **4**
- 2.1 Library Files Description . . . 4
- 2.2 Gradient Checking . . . 5

3 XOR Problem **5**
- 3.1 Implementation . . . 5
- 3.2 Results . . . 5

4 Autoencoder for MNIST Reconstruction **6**
- 4.1 Importance . . . 6
- 4.2 Architecture . . . 6
- 4.3 Subsampling for Efficient Training . . . 6
- 4.4 Reconstruction Results . . . 6

5 Latent Space Classification **7**
- 5.1 Classification Results . . . 7
- 5.2 Analysis . . . 8

6 TensorFlow/Keras Comparison **8**
- 6.1 XOR Comparison . . . 8
- 6.2 Auto Encoder Comparison . . . 9
- 6.3 General Conclusions . . . 10

7 Implementation Insights and Lessons Learned **10**

8 Conclusion **10**

List of Figures

1 Training loss for the XOR problem . . . 5
2 Decision Boundary for the XOR problem . . . 6
3 Autoencoder reconstruction loss . . . 7
4 Original images (top) and reconstructed images (bottom) . . . 7
5 Confusion matrix for SVM classification . . . 8
6 Decision Boundary for the XOR problem with TensorFlow . . . 9
7 Comparison between our library and TensorFlow . . . 9

List of Tables

1 Autoencoder Performance Comparison . . . 10

1 Introduction

This project focuses on building a complete neural network library from scratch using Python and NumPy, with the goal of deeply understanding the internal mechanics of modern machine learning systems. Instead of relying on high-level frameworks, all core components such as layers, activation functions, loss functions, optimizers, and training loops are implemented manually.
The developed library is validated through multiple experiments including gradient checking, solving the XOR problem, training an autoencoder on the MNIST dataset, and performing digit classification using a Support Vector Machine (SVM) on learned latent representations. Finally, results are compared against TensorFlow/Keras implementations to evaluate performance, training behavior, and ease of use.

2 Library Design and Architecture

The neural network library was designed in a modular and extensible manner. Core components such as layers, activation functions, loss functions, optimizers, and the training loop are implemented as independent modules.
Each layer implements both forward and backward passes, enabling automatic gradient propagation. This design allows flexibility in constructing networks while ensuring clarity and maintainability. The library relies solely on NumPy to provide full transparency into the learning process.
The project repository is organized as follows:

```
lib/
 |---- __init__.py
 |---- activations.py
 |---- losses.py
 |---- layers.py
 |---- optimizers.py
 |---- network.py
 |---- preprocessing.py
 |---- svm.py
 \---- visualize.py
notebooks/
 |---- project_demo.ipynb
 \---- weights/
     |---- encoder_weights.npz
     |---- decoder_weights.npz
     \---- svm_weights.npz
```

2.1 Library Files Description

- **__init__.py**: Initializes the library and allows importing modules as a package.
- **layers.py**: Implements neural network layers such as fully connected (Dense) layers, including forward and backward propagation logic.
- **activations.py**: Contains activation functions such as ReLU, Sigmoid, and Tanh, along with their analytical gradients for backpropagation.
- **losses.py**: Implements loss functions including Mean Squared Error (MSE) used for regression and autoencoder reconstruction.
- **optimizers.py**: Contains optimization algorithms such as Stochastic Gradient Descent (SGD) used to update model parameters during training.
- **network.py**: Defines the core Network class responsible for chaining layers, performing forward and backward passes, managing training loops, and saving/loading model weights.

- **preprocessing.py**: Handles dataset loading, normalization, reshaping, and subsampling operations.
- **svm.py**: Implements a multi-class Support Vector Machine classifier from scratch, trained using gradient descent on hinge loss.
- **visualize.py**: Contains utilities for plotting loss curves, decision boundaries, reconstruction results, and confusion matrices.
- **project_demo.ipynb**: A single notebook demonstrating all experiments, including training, evaluation, and visualization.
- **weights/**: Stores saved model parameters in `.npz` format to avoid retraining models on every run.

2.2 Gradient Checking

To verify the correctness of backpropagation, numerical gradient checking was performed using finite differences:

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

The analytical gradients closely matched numerical gradients, confirming the correctness of the implementation, as shown:

```
--- Layer 0 ---
Max difference in W: 3.96e-13
Max difference in b: 7.91e-13
Gradients match within tolerance.

--- Layer 2 ---
Max difference in W: 5.96e-13
Max difference in b: 6.47e-13
Gradients match within tolerance.
```

3 XOR Problem

3.1 Implementation

The XOR problem was implemented using a fully connected neural network with a hidden layer and non-linear activation functions. This configuration enables the model to learn a non-linearly separable decision boundary.

3.2 Results

The network successfully learned the XOR mapping.

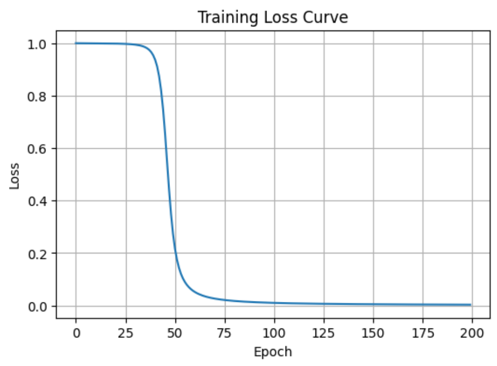


Figure 1: Training loss for the XOR problem

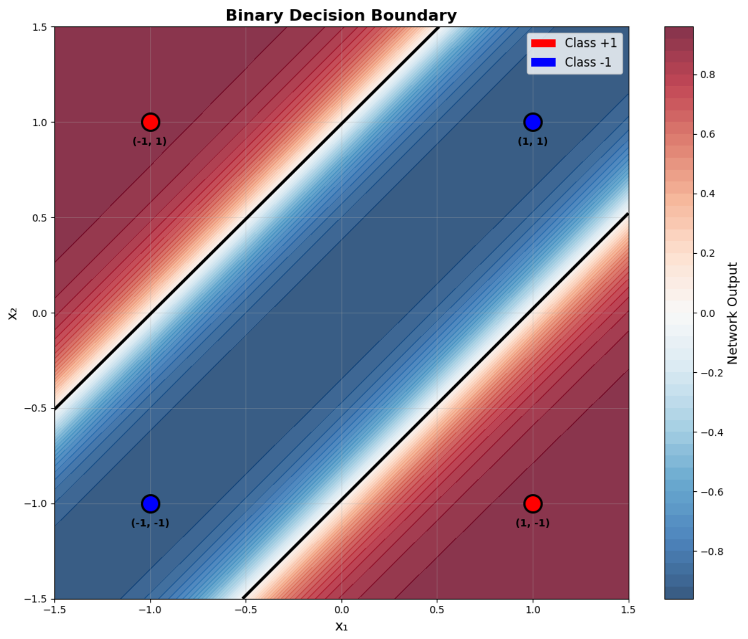


Figure 2: Decision Boundary for the XOR problem

4 Autoencoder for MNIST Reconstruction

4.1 Importance

Autoencoders are unsupervised neural networks designed to learn compact and meaningful representations of data. By constraining the latent space dimensionality, the model is forced to extract salient features such as edges, shapes, and digit structure.

4.2 Architecture

The autoencoder consists of:

- Encoder: Dense layers compressing 784-dimensional inputs into a low-dimensional latent space.
- Decoder: Dense layers reconstructing the original image from the latent representation.

4.3 Subsampling for Efficient Training

To reduce training time while preserving representativeness, a subset of the MNIST dataset was used:

- Training samples: 5000
- Test samples: 1000

4.4 Reconstruction Results

The autoencoder successfully reconstructs digit shapes while losing fine-grained details due to compression. Figures below show the training loss curve and reconstruction results on the test dataset.

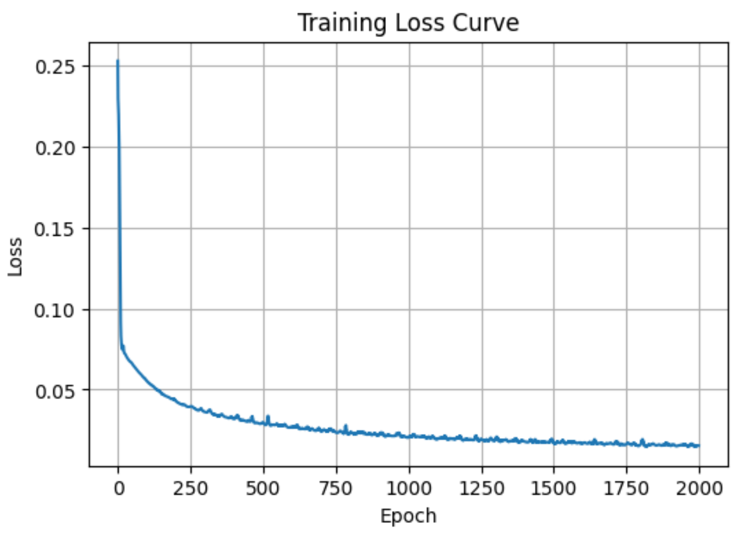


Figure 3: Autoencoder reconstruction loss

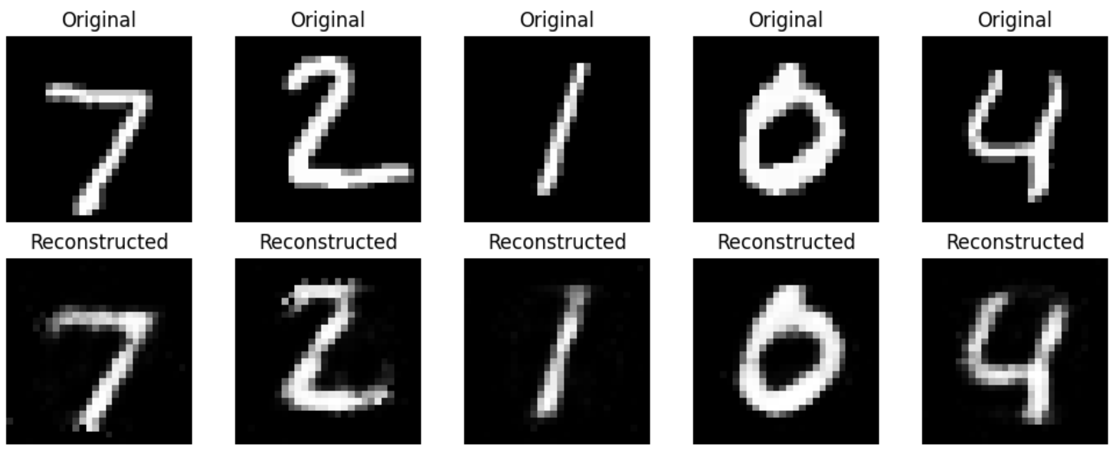


Figure 4: Original images (top) and reconstructed images (bottom)

5 Latent Space Classification

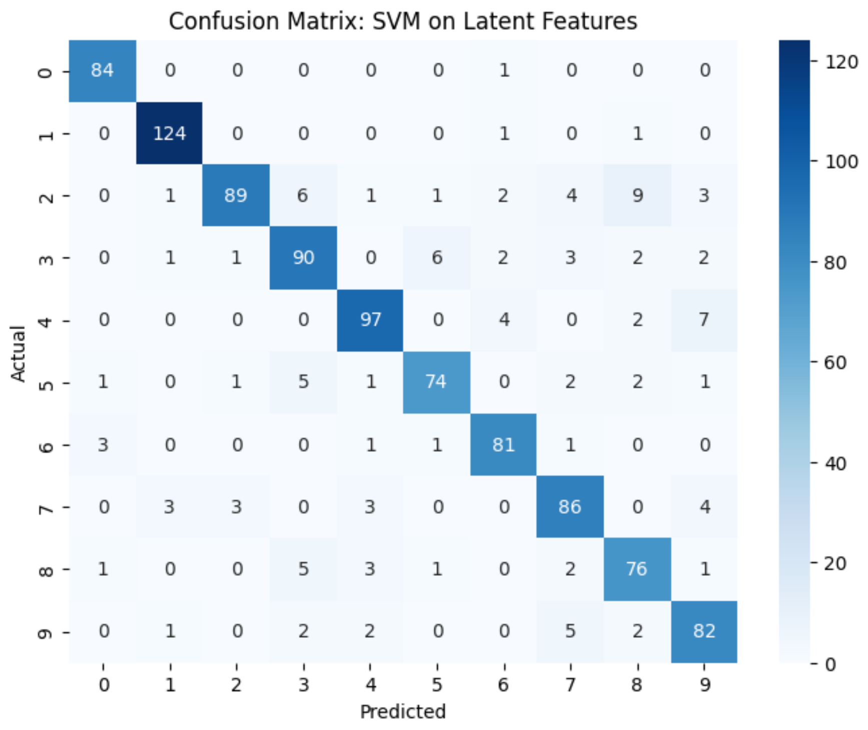
After training the autoencoder, only the encoder part was retained and used as a feature extractor. Each MNIST image was transformed into a compact latent vector, which served as input to the custom SVM classifier.

5.1 Classification Results

The SVM achieved an accuracy of **88.3%** on the test set.

```
SVM Accuracy: 88.30%

Classification Report:
Class 0: precision=0.94, recall=0.99, f1-score=0.97, support=85
Class 1: precision=0.95, recall=0.98, f1-score=0.97, support=126
Class 2: precision=0.95, recall=0.77, f1-score=0.85, support=116
Class 3: precision=0.83, recall=0.84, f1-score=0.84, support=107
Class 4: precision=0.90, recall=0.88, f1-score=0.89, support=110
Class 5: precision=0.89, recall=0.85, f1-score=0.87, support=87
Class 6: precision=0.89, recall=0.93, f1-score=0.91, support=87
Class 7: precision=0.83, recall=0.87, f1-score=0.85, support=99
Class 8: precision=0.81, recall=0.85, f1-score=0.83, support=89
Class 9: precision=0.82, recall=0.87, f1-score=0.85, support=94
```



Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	84	0	0	0	0	0	1	0	0	0
1	0	124	0	0	0	0	1	0	1	0
2	0	1	89	6	1	1	2	4	9	3
3	0	1	1	90	0	6	2	3	2	2
4	0	0	0	0	97	0	4	0	2	7
5	1	0	1	5	1	74	0	2	2	1
6	3	0	0	0	1	1	81	1	0	0
7	0	3	3	0	3	0	0	86	0	4
8	1	0	0	5	3	1	0	2	76	1
9	0	1	0	2	2	0	0	5	2	82

Figure 5: Confusion matrix for SVM classification

5.2 Analysis

Digits with distinct shapes achieved higher precision and recall, while visually similar digits exhibited higher confusion. This indicates that the latent space captures meaningful global features but still contains overlapping class regions.

6 TensorFlow/Keras Comparison

The same architectures were implemented using TensorFlow/Keras. The TensorFlow models converged faster and achieved slightly lower reconstruction loss due to optimized computation. However, the learning behavior and qualitative results were consistent with the custom implementation.

6.1 XOR Comparison

The following results were obtained when applying the same model to the XOR problem.

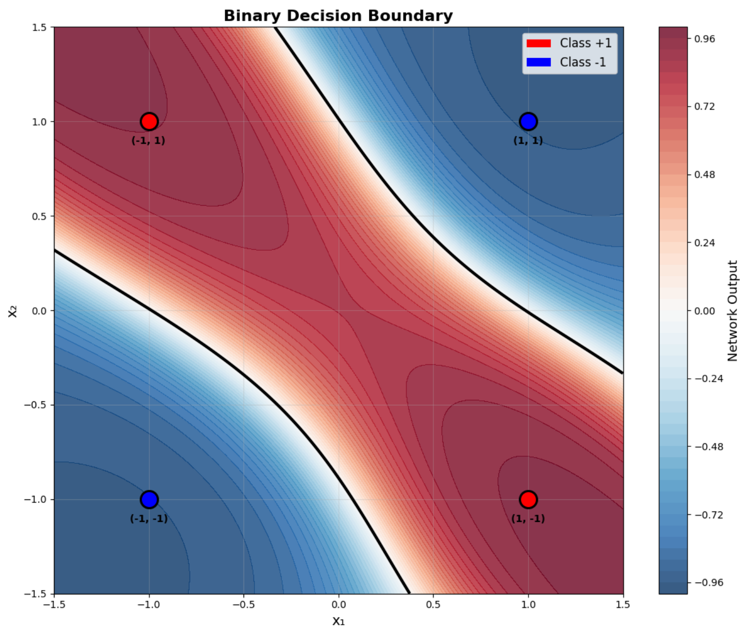


Figure 6: Decision Boundary for the XOR problem with TensorFlow

Performance Comparison:

Aspect	Custom Model	TensorFlow Model
Training Time	Less	Much More
Ease of Implementation	Harder	Easier
Flexibility	High	Medium
Debuggability	Easier	Slightly Complex
Integration	Manual	Built-in

6.2 Auto Encoder Comparison

The following results were obtained when applying the same model to the Auto Encoder problem.

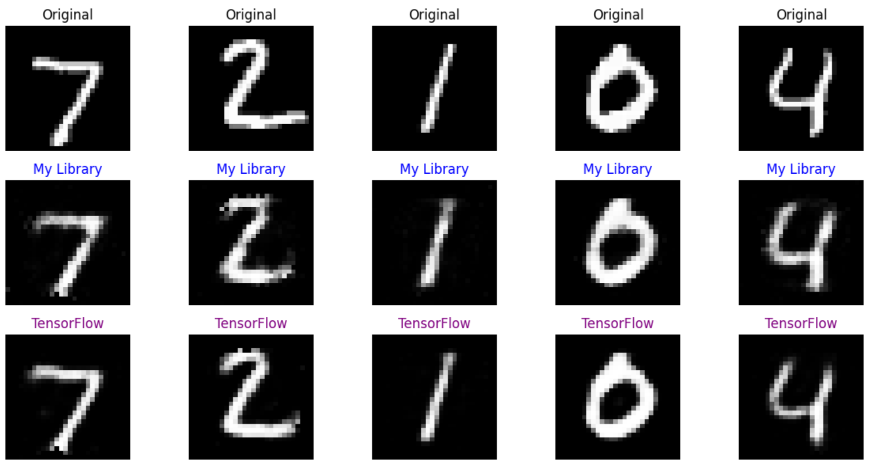


Figure 7: Comparison between our library and TensorFlow

Performance Comparison:

Table 1: Autoencoder Performance Comparison

Metric	**Custom Library**	**TensorFlow/Keras**
Final Loss (MSE)	0.015425	0.005318
Training Time (minutes)	24.76	11.37

6.3 General Conclusions

The custom implementation provides full control and transparency, making it ideal for learning and experimentation. TensorFlow, on the other hand, enables rapid development and optimized performance, making it more suitable for large-scale applications.

7 Implementation Insights and Lessons Learned

- Implementing the neural network library from scratch improved understanding of forward propagation, backpropagation, and gradient flow across layers.
- Numerical gradient checking validated the correctness of the analytical gradients and increased confidence in the training pipeline.
- Proper weight initialization and input scaling were essential for stable convergence and avoiding numerical issues.
- Model architecture depth had a clear impact: shallow networks were sufficient for XOR, while deeper encoder–decoder structures were required for meaningful MNIST representations.
- Latent features learned by the autoencoder enabled effective SVM classification, demonstrating the value of unsupervised feature learning.
- Comparing the custom implementation with TensorFlow highlighted the trade-off between full control and ease of use provided by high-level frameworks.

8 Conclusion

This project demonstrates a complete machine learning pipeline built from first principles, including neural network training, unsupervised representation learning, and downstream classification using classical machine learning techniques.