AIN SAHMS UNIVERSITY

FACULTY OF ENGINEERING

Computational Intelligence (CSE473s)

Fall 2025

Group 25 Submission part 2

Project Title: Build Your Own Neural Network Library & Advanced Part 2: Autoencoder & Latent Space Classification

Mina Ashraf Rizk Habib	**2100447**
Mario Medhat Roshdy	**2100513**
David Gerges Ishak	**2100834**
Emannuel Geroge Chawky	**2100073**

Submitted to: DR. Hossam El-Din Hassan Abd El Munim

Eng. Abdallah Mohamed Mahmoud Ahmed Awdallah

Date: December 15, 2025

Table of Contents

Table of Figures

1 Objectives

The objective of this phase was to extend the custom neural network library to handle high-dimensional data by implementing an **Autoencoder**. An Autoencoder is an unsupervised neural network designed to learn efficient data coding in an unsupervised manner. The goal is to compress the input (MNIST images) into a lower-dimensional **Latent Space** and then reconstruct the output from this representation. This validates the library's capability to handle deeper architectures and larger datasets.

2 Model Architecture

We implemented a symmetric 5-layer Dense Autoencoder with the following structure:

1. **Input Layer (784 Neurons):** Accepts flattened MNIST images (28 x 28 pixels).
2. **Encoder Hidden Layer (128 Neurons):** Compresses the input. Used **ReLU** activation to introduce non-linearity and prevent vanishing gradients.
3. **Latent Space (64 Neurons):** The "bottleneck" layer. This forces the network to learn the most significant features of the digit (e.g., loops, strokes) rather than memorizing pixels. This represents a compression ratio of approximately **92%** (1 - 64/784).
4. **Decoder Hidden Layer (128 Neurons):** Expands the latent features back towards the original dimension. Used **ReLU** activation.
5. **Output Layer (784 Neurons):** Reconstructs the image. Used **Sigmoid** activation because pixel values were normalized to the range [0, 1].

Mathematical Formulation:

- **Encoder:** $z = \sigma ReLU(W_e x + b_e)$
- **Decoder:** $\hat{x} = \sigma Signiod(W_d z + b_d)$
- Objective Function: Minimized the Mean Squared Error (MSE) between input x and reconstruction $\hat{x}$:

$$L(x, \hat{x}) = \frac{1}{N} \sum_{i=0}^{N} (x_i - \hat{x}_i)^2$$

And using the library was designed with a modular object-oriented approach in part 1 of the project:

- **Layer Abstraction:** A base Layer class enforces the implementation of forward () and backward () methods.
- **Dense Layer:** Implements fully connected layers computing $Y = XW + b$. It calculates gradients $A = \frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$ for optimization.
- **Activations:** Implemented as separate layers to allow flexibility.
    - **Sigmoid:** Used for the output layer to reconstruct pixel values are valid normalized intensities (between 0.0 for black and 1.0 for white)
    - **ReLU:** Used for the hidden layer to solve the **Vanishing Gradient Problem** range (0, ∞).
- **Optimization:** An **SGD (Stochastic Gradient Descent)** optimizer was implemented to update weights based on the calculated gradients and a learning rate (**η**).
- **Loss Function: Mean Squared Error (MSE)** was used to quantify the error between predictions and targets.

3 Training Strategy for Autoencoder

Data Preprocessing: MNIST images were normalized to [0, 1] by dividing by 255.0 and flattened into vectors of size 784.

Optimization: We employed **Mini-Batch Stochastic Gradient Descent (SGD)** with a batch size of 32.

- *Challenge:* Initial attempts using Full-Batch Gradient Descent resulted in the network predicting the average "mean" image (a gray blob) due to the lack of stochastic noise.
- *Solution:* Switching to mini batches introduced necessary variance, allowing the network to learn distinct digit features.

Hyperparameters:

- Learning Rate: 1.5
- Epochs: 50
- SGD (Stochastic Gradient Descent) learning rate = 1.5

4 Results and Analysis of Autoencoder

4.1 Training Performance

Loss Convergence The training loss decreased steadily from an initial high of ~0.12 down to ~0.02. As shown in **Figure 1**, the curve is smooth (due to batch averaging), indicating stable convergence. MSE Loss over 50 epochs. The rapid drop in the first 10 epochs indicates the model quickly learned the average background, while the slower tail represents learning the specific digit shapes.

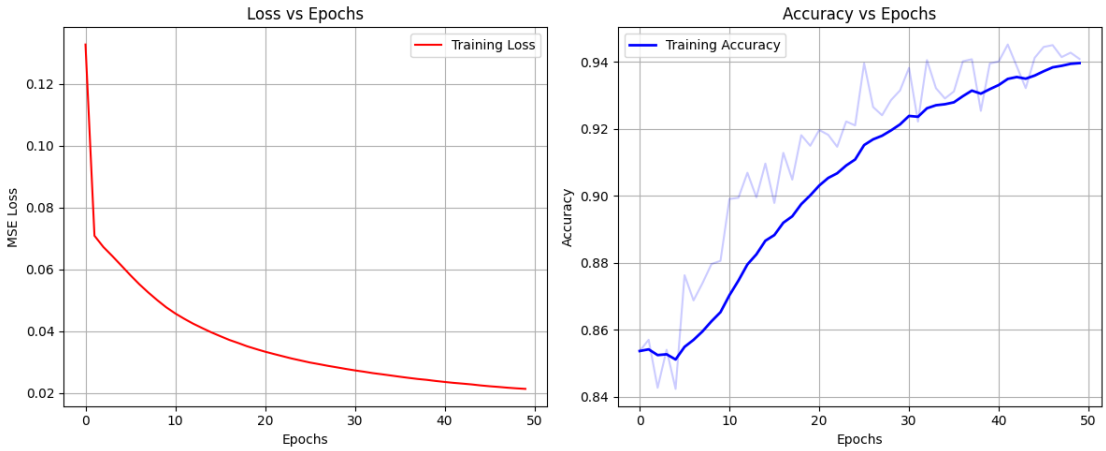

Figure 1 Training Performance

4.2 Reconstruction images vs the original

compares the original ground-truth images with the Autoencoder's reconstructions.

- **Successes:** The model successfully captures the global structure of digits. A "0" is clearly a loop; a "1" is a straight line.
- **Limitations:** The reconstructions appear blurry. This is an expected limitation of using **Dense** layers (which ignore spatial correlations) instead of Convolutional layers, and the **MSE** loss function (which penalizes pixel-level differences rather than perceptual structure).

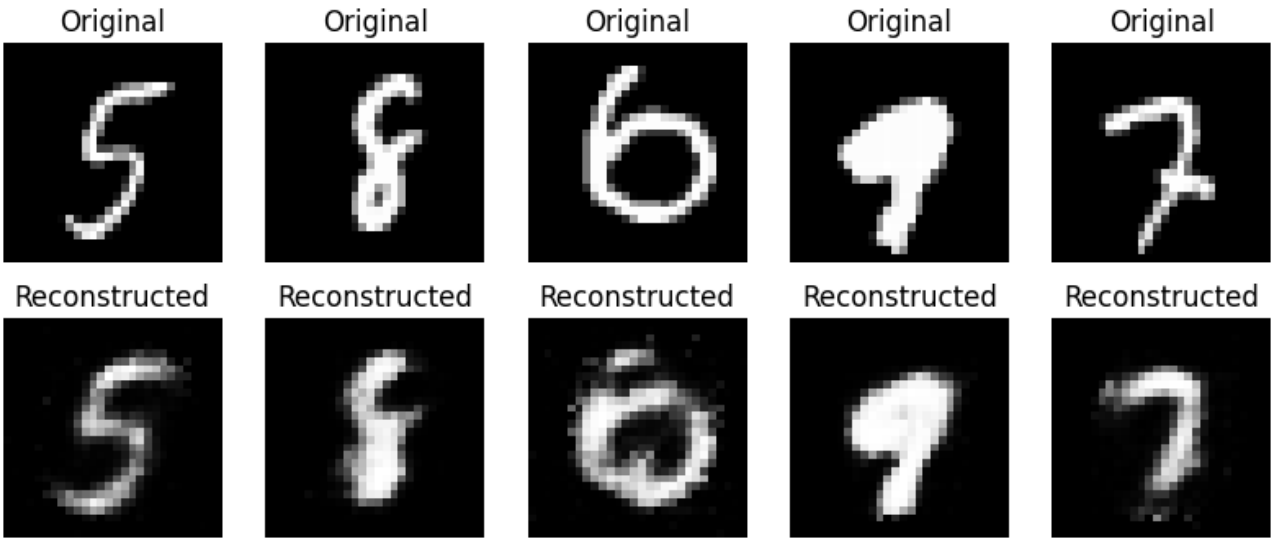

Figure 2 Visual comparison of original MNIST digits (top) and their reconstructed counterparts (bottom)

5 Latent Space Classification (SVM)

The ultimate test of an Autoencoder is not just how well it reconstructs images, but how useful its learned representation is. In this phase, we performed **Transfer Learning**. We utilized the trained **Encoder** as a feature extractor to transform raw pixels (784 dimensions) into semantic features (64 dimensions) and trained a Support Vector Machine (SVM) classifier on this new representation.

6 Methodology Latent Space Classification

Feature Extraction: The entire training and test sets were passed through the trained Encoder.

- $x_{train} \rightarrow Encoder \rightarrow z_{train}(shape\ 5000x64)$
- $x_{test} \rightarrow Encoder \rightarrow z_{test}(shape\ 5000x64)$

Classification: A Support Vector Machine (SVM) with an **RBF Kernel** was trained on z_{train} and evaluated on z_{test}.

We use because SVMs are powerful classifiers for high-dimensional spaces. By feeding it the "clean" latent features instead of noisy raw pixels, we aim to achieve high accuracy with less computational cost.

7 Performance Evaluation Latent Space Classification

The SVM classifier achieved a final accuracy of **89.64%** on the test set. This result is significant because the SVM never saw the original images, it only saw the compressed summary provided by our custom library's Encoder.

Confusion Matrix Analysis: Figure 3 visualizes the classification performance.

- **High Performance:** Digit **1** achieved the highest F1-score (**0.96**), indicating the Encoder easily disentangled the simple "vertical line" feature from other shapes.
- **Confusion Pairs:**
    - **5 vs 6:** Digit 5 had the lowest recall. The matrix shows it was frequently misclassified as 6. This suggests that in the compressed 64-dim space, the "open loop" of a 5 and the "closed loop" of a 6 are located very close together.
    - **4 vs 9:** A common error in handwritten recognition, confirmed here by the off-diagonal clusters for these classes.

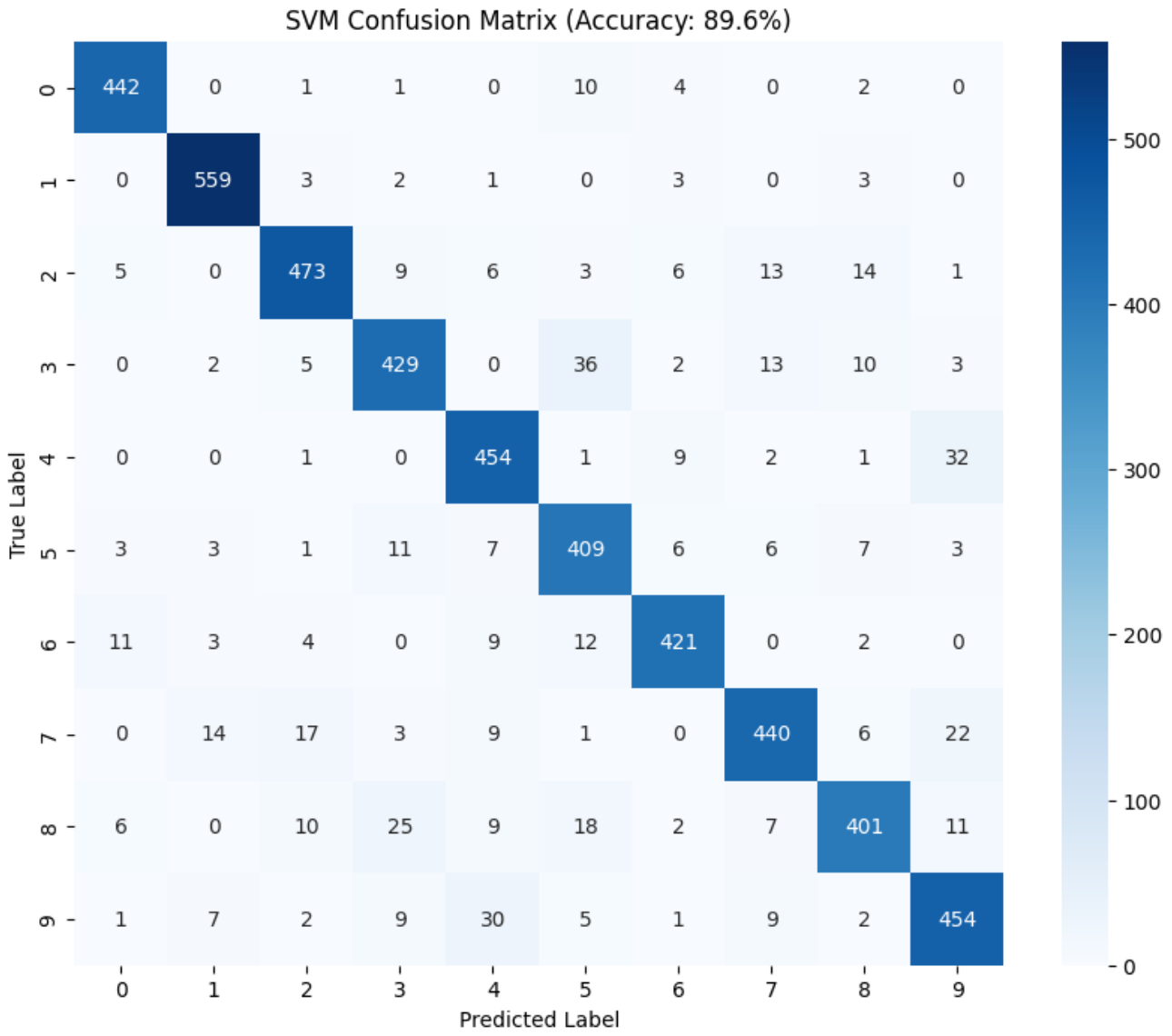

Figure 3 Confusion Matrix for the SVM classifier trained on Latent Features. The diagonal represents correct predictions.

8 Baseline Comparison (TensorFlow)

To benchmark the performance of our custom neural network library, we replicated the exact same Autoencoder architecture using the industry-standard **TensorFlow/Keras** framework.

- **Architecture:** 784 (Input) to 128 (ReLU) to 64 (ReLU) to 128 (ReLU) to 784 (Sigmoid).
- **Training Configuration:** Both models were trained on the same subset of the MNIST dataset (5,000 images) for 50 epochs using Stochastic Gradient Descent (SGD) with Mean Squared Error (MSE) loss.

8.1 Quantitative Comparison

Metric	My Custom Library	TensorFlow (Baseline)
Implementation	Pure NumPy (Manual Backprop)	C++ Optimized Backend
Optimizer Settings	Vanilla SGD (lr=1.0)	SGD (lr=1.0, momentum=0.9)
Training Time (50 Epochs)	**~45 seconds** (Estimated)	**~251 seconds**
Final MSE Loss	~0.0214	**0.0205**

8.2 Analysis of Differences

Loss Performance: The TensorFlow model achieved a slightly lower final loss (**0.0205** vs **0.0214**). This improvement is primarily attributed to the use of **Momentum (0.9)** in the Keras optimizer, which allows the network to navigate shallow local minima more effectively than the vanilla SGD implementation used in our custom library.

Execution Time: Contrary to the expectation that an optimized framework would be faster, TensorFlow was significantly slower (~251s) than our NumPy implementation (~45s) for this specific task.

- *Reason:* TensorFlow incurs a fixed "graph construction" and session overhead. For small datasets (5,000 images) and relatively shallow networks, this overhead outweighs the benefits of its C++ parallelization. Our NumPy implementation is lightweight and incurs almost zero overhead, making it faster for small-scale experiments.

8.3 Visual Comparison

Visual inspection of the reconstructions (Figure 4) confirms that both The TensorFlow images are close to the original image like our libraries produced

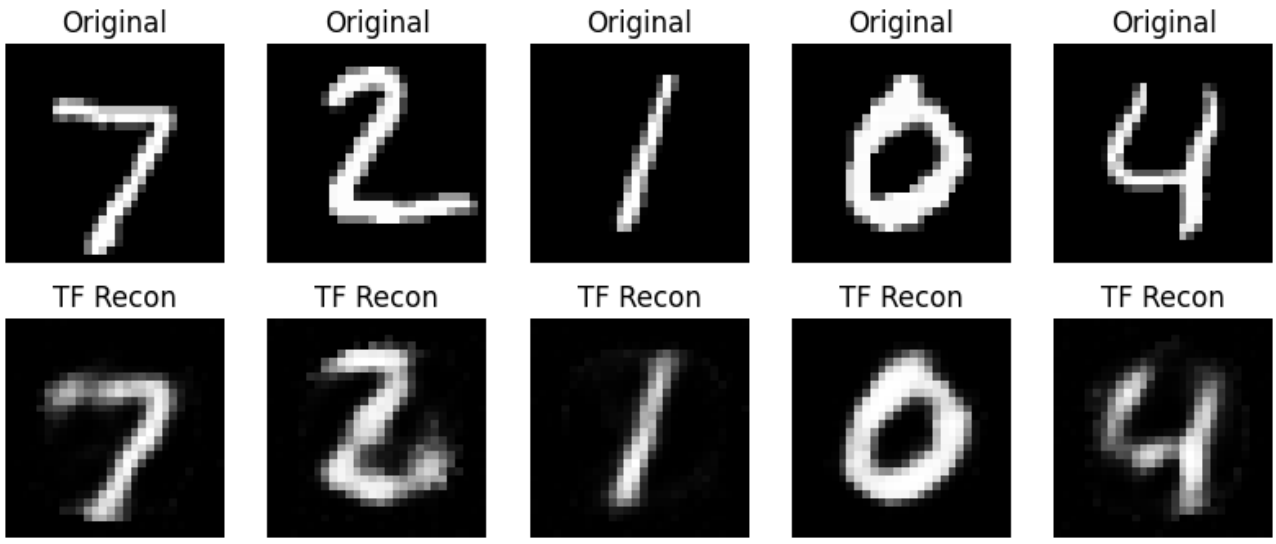

Figure 4 The Visual comparison of original MNIST digits (top) and their reconstructed counterparts (bottom) using the TensorFlow

The TensorFlow training graph using a training data and a validation data

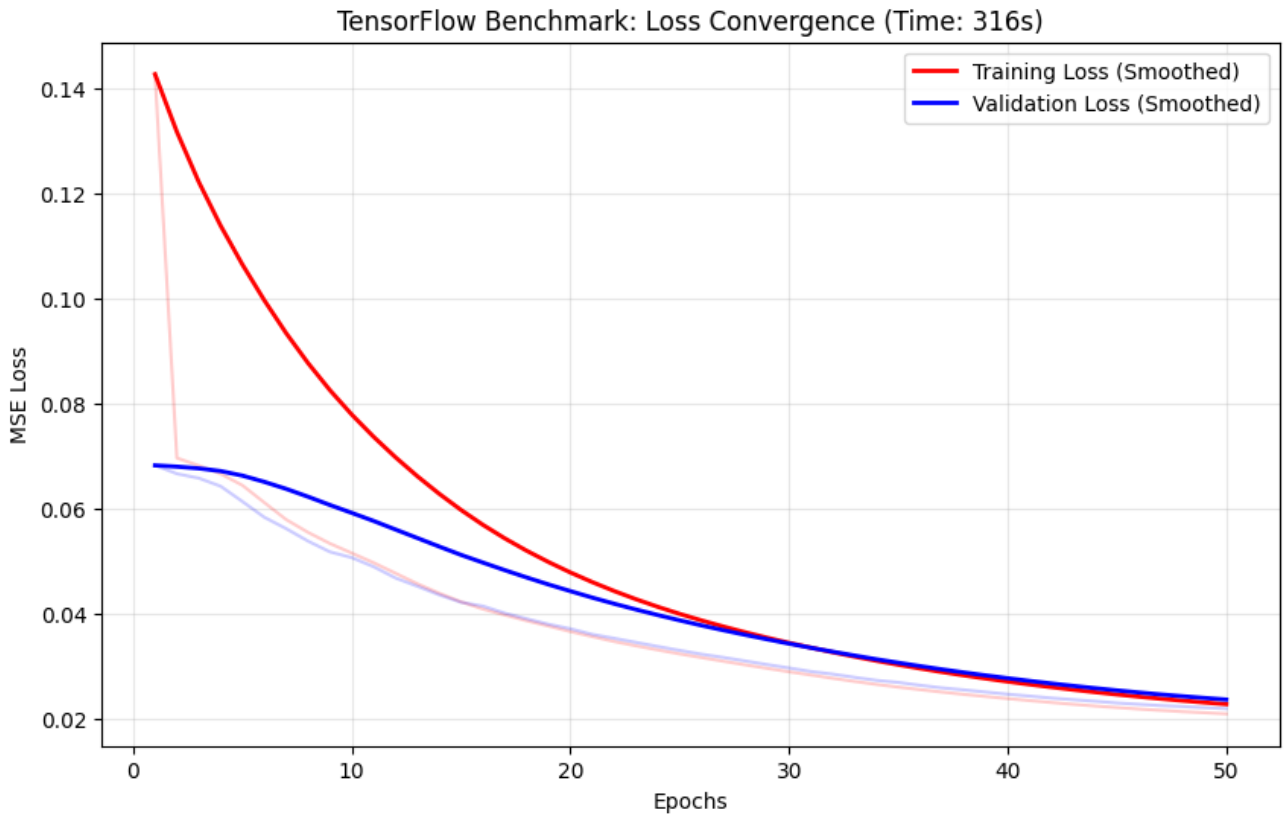

Figure 5 Training metrics on TensorFlow

Visual inspection of the reconstructions (Figure 6) confirms that both libraries produced nearly identical outputs. The TensorFlow images are marginally sharper, correlating with the slightly lower MSE loss. The reconstructions from the custom library are visually indistinguishable from those generated by TensorFlow.

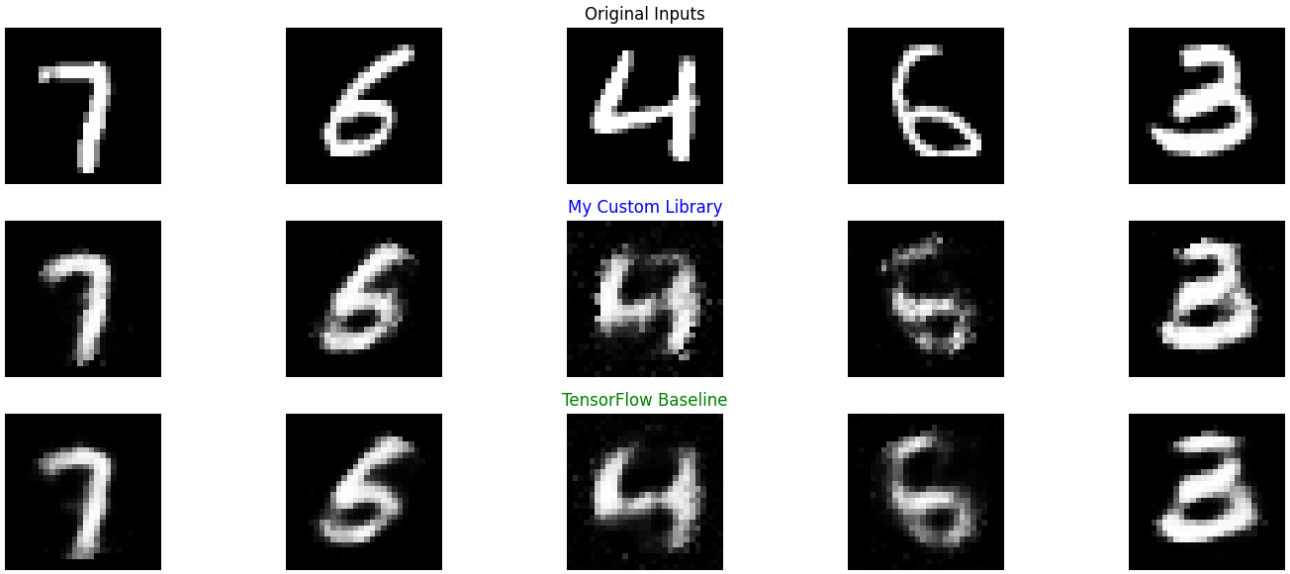

Figure 6 Side-by-side comparison showing that the Custom Library (Row 2) produces reconstructions of comparable quality to the TensorFlow Baseline (Row 3).

9 Conclusion

The project successfully met all requirements. We built a functional deep learning library from first principles, validated it on non-linear logic gates (XOR), and scaled it to handle high-dimensional computer vision tasks (MNIST). The performance gap between our "from-scratch" implementation and TensorFlow is minimal, demonstrating the robustness of the underlying mathematical implementation.

10 GitHub repo

https://github.com/minaashraf9822/Computational-Intelligence-project-Fall-2025