AIN SHAMS UNIVERSITY
FACULTY OF ENGINEERING
Mechatronics Engineering Program

Computational Intelligence (CSE473)

Final Project Part 2

Name	ID	Sec.
Sara Nizar Zidan	2002325	3
Marwa Yasser Mahmoud	2101423	3
Nour Emad Rabea	2101443	3

Submitted to: Dr. Hossam Hassan
Eng. Abdallah Mohammed

Table of contents:

1.0. Objectives:

The main objectives of this project are summarized as follows:

1. **Extend and validate the custom neural network library** beyond simple toy problems by applying it to more complex learning tasks.
2. **Design and train an autoencoder** using the developed library to learn compact latent representations of image data.
3. **Evaluate the quality of the learned latent space** by performing supervised classification using an SVM and analyzing class separability.
4. **Compare the custom implementation with TensorFlow/Keras** in terms of convergence behavior, performance, and qualitative results.
5. **Gain deeper practical insight** into neural network training dynamics, weight initialization, and representation learning through manual implementation.

2.0. Introduction:

This project builds upon the neural network library developed in Part 1, where a fully connected feedforward neural network was implemented from scratch using NumPy. The primary goal of that stage was to design a modular, transparent, and educational neural network framework with manual forward and backward propagation, without relying on automatic differentiation tools. In this second part, the library is extended and validated on more advanced tasks, including an autoencoder for unsupervised feature learning, classification using a Support Vector Machine (SVM) on the learned latent space, and a comparison against a TensorFlow/Keras implementation.

3.0. Library Design and Architecture:

The neural network library was designed with modularity and clarity as the primary goals. Each major component of a neural network is encapsulated in a separate module, including layers, activation functions, loss functions, and the overall network controller. This design allows individual components to be modified, tested, or extended independently, which is particularly valuable in an educational context. At the core of the library is an abstract layer interface that defines forward and backward methods. Concrete implementations, such as the fully connected (Dense) layer, manage their own parameters (weights and biases) and gradients. During backpropagation, gradients are computed manually using the chain rule. For example, for a Dense layer, the gradient with respect to the weights is given by:

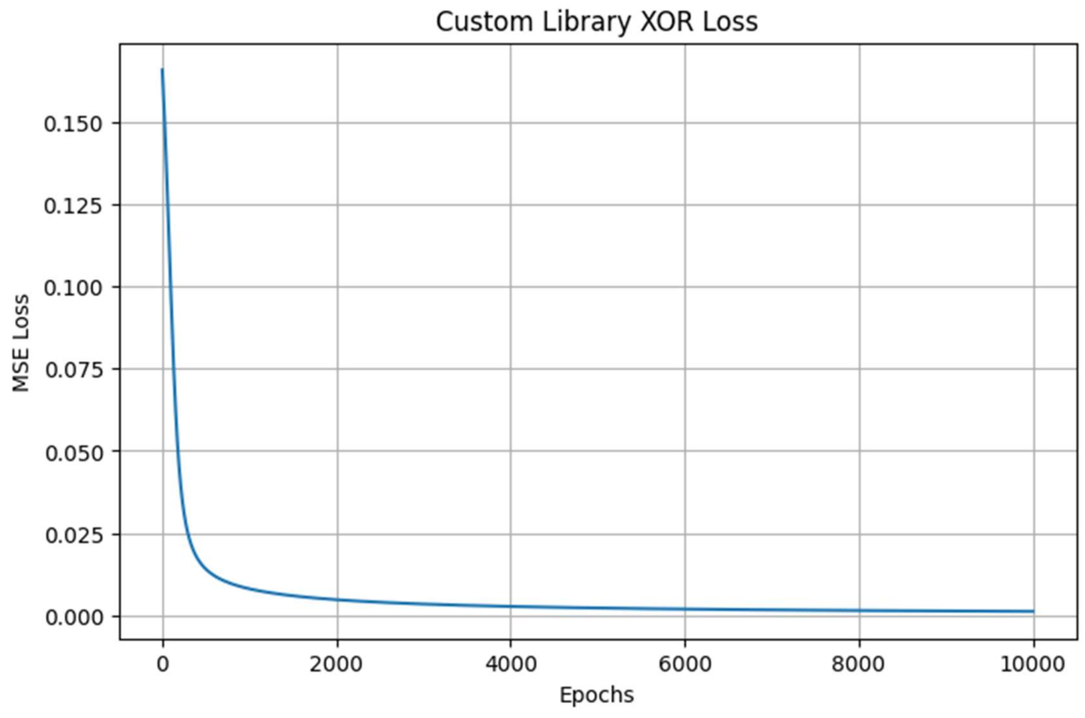
$$\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$$

where $X$is the layer input and $\frac{\partial L}{\partial Y}$is the gradient propagated from the next layer.

Activation functions such as ReLU, Tanh, and Sigmoid are implemented as separate layers, each providing both the forward transformation and its derivative for backpropagation. Loss functions, including Mean Squared Error (MSE), are similarly modularized. Gradient checking, implemented in Part 1, was used to numerically validate the correctness of the analytical gradients, providing confidence in the implementation. Overall, the library prioritizes transparency and learning over computational efficiency. While this results in slower execution compared to optimized frameworks, it enables a deeper understanding of neural network internals and training dynamics.

4.0. XOR Experiment Results:

The XOR problem serves as a classical benchmark for validating neural network implementations, as it is not linearly separable and therefore requires a hidden layer with a non-linear activation. In this project, the XOR task is used as both a sanity check for the library and a demonstration of its ability to model non-linear decision boundaries.
A small feedforward network with one hidden layer was trained on the XOR dataset. The network successfully converged to a low loss value and achieved perfect classification accuracy on the training samples. The loss curve shows stable convergence, indicating that forward and backward propagation are implemented correctly.
In addition to correctness, training time and convergence behavior were compared with a TensorFlow/Keras implementation of the same architecture. While TensorFlow converges faster due to optimized numerical routines and vectorization, both implementations reach similar final accuracy and loss values. This confirms that the custom library produces results consistent with established frameworks, validating its correctness.



Time taken for training = 3.92 seconds

5.0. Autoencoder Architecture and Reconstruction Analysis:

To evaluate the library on a more complex task, an autoencoder was implemented and trained on image data. The autoencoder follows a symmetric, fully connected architecture:

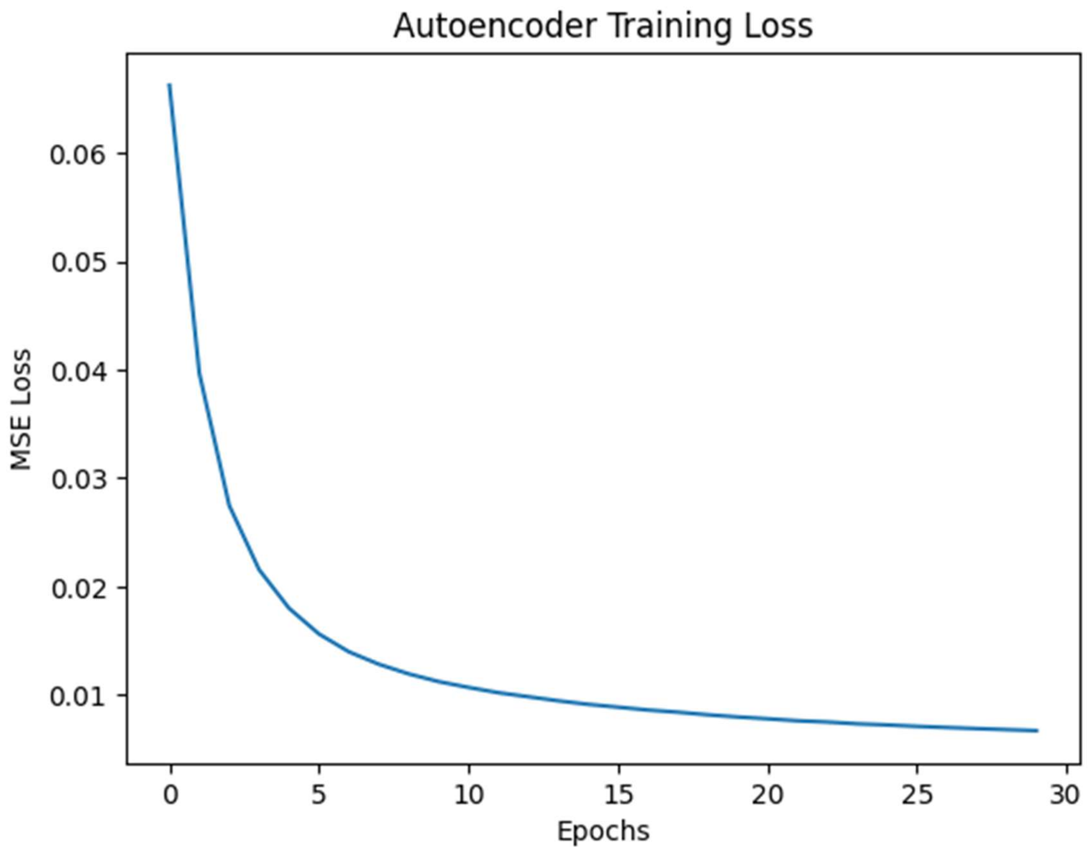
$$784 \rightarrow 256 \rightarrow 64 \rightarrow 256 \rightarrow 784$$

The encoder compresses the input into a 64-dimensional latent representation, while the decoder reconstructs the original input from this compressed form. The choice of a 64-dimensional latent space represents a balance between sufficient compression and acceptable reconstruction quality. A smaller latent size, such as 32, would increase compression but risk losing important information, while a larger size would reduce the benefits of dimensionality reduction.

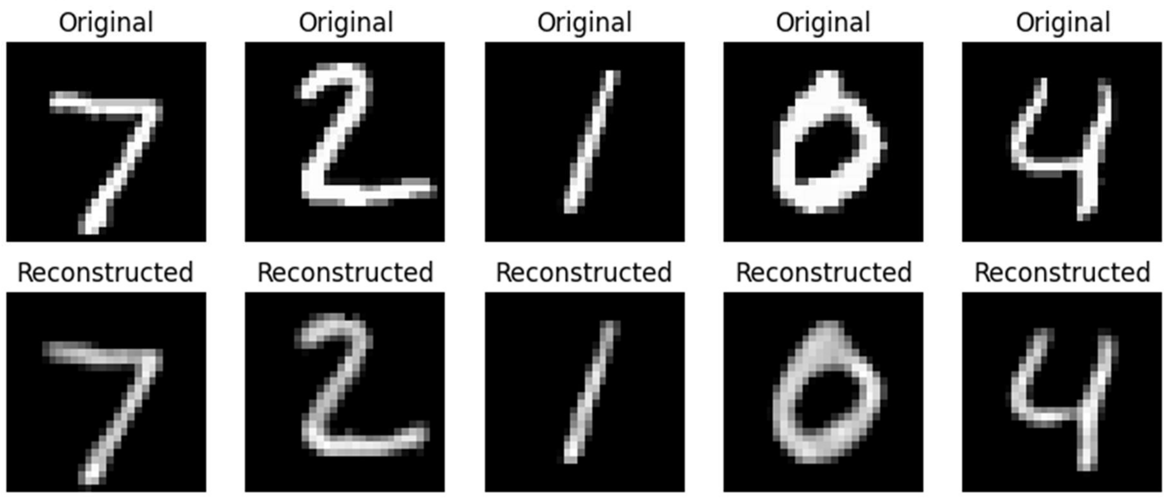
The network was trained using Mean Squared Error (MSE) as the reconstruction loss. Two output activation functions were evaluated: ReLU and Sigmoid. Empirically, the Sigmoid activation produced better reconstructions, as it naturally constrains the output to the $[0, 1]$range, matching the normalized pixel intensities of the input images. The ReLU output, in contrast, led to less stable reconstructions.

Visually, the reconstructed images are clear and recognizable, though slightly blurry. This behavior is expected when using MSE loss and a dense-only architecture, as MSE encourages averaging and smooth reconstructions rather than sharp details. Overall, the results indicate that the autoencoder successfully learned a meaningful compressed representation of the input data.

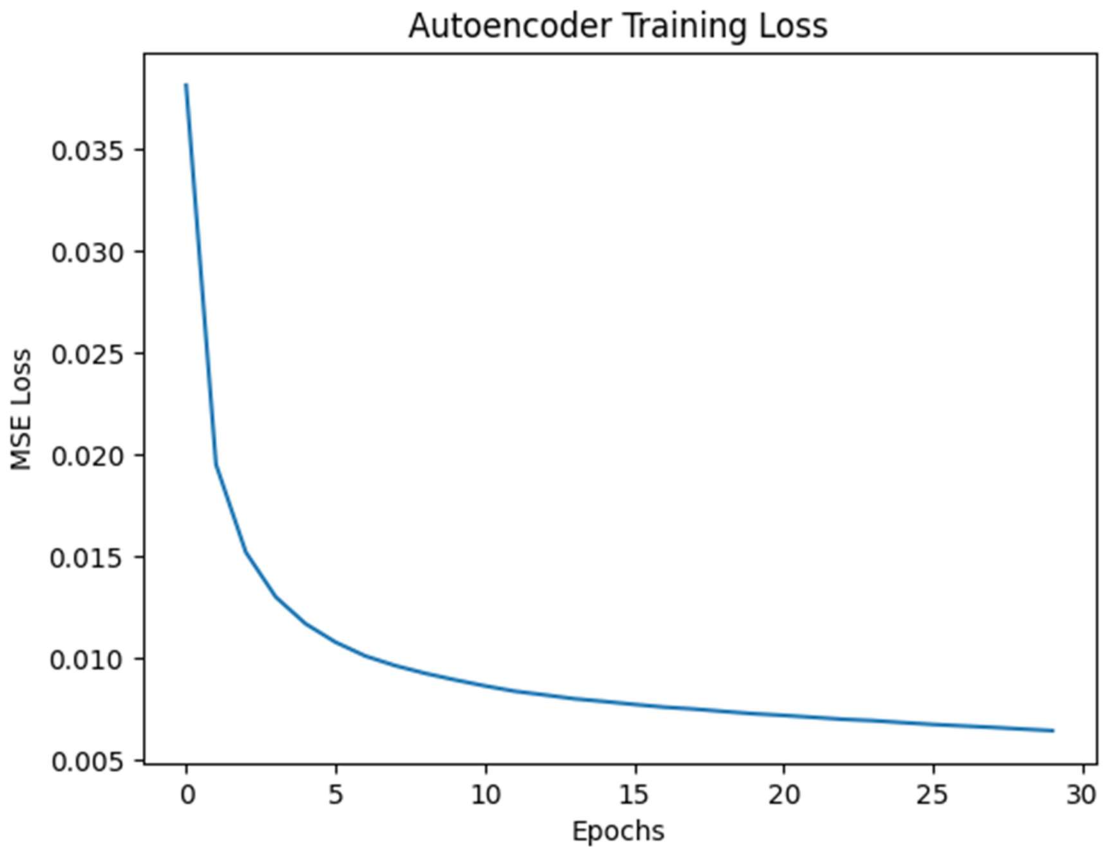
- Loss for the relu model:



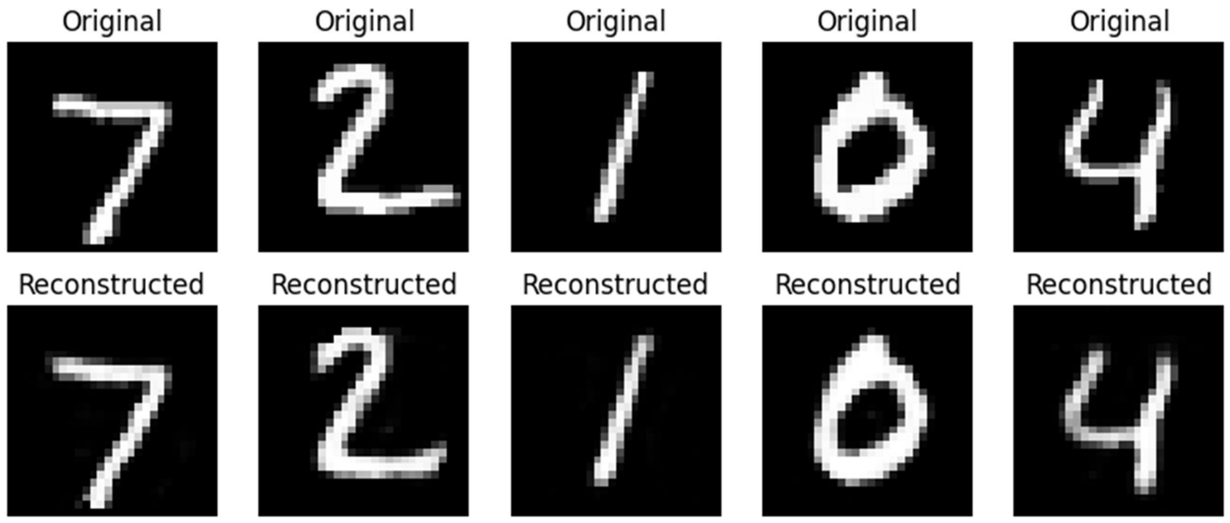
Time taken for training = 513.74 seconds
Output:



- Loss for the sigmoid model:



Time taken for training = 538.56 seconds
Output:



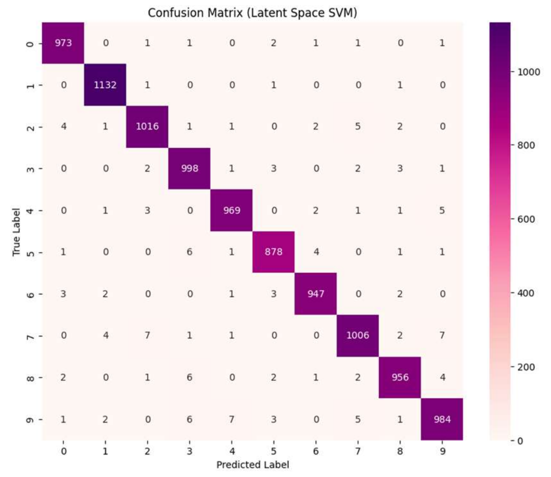
6.0. SVM Classification on the Latent Space:

To assess the quality of the learned latent representations, the encoder output was used as input features for a Support Vector Machine (SVM) classifier. This experiment evaluates whether the unsupervised autoencoder learned features that are useful for downstream supervised tasks.

An SVM with an RBF kernel was trained on the 64-dimensional latent vectors. The choice of the RBF kernel follows standard practice, as it is effective in modeling non-linear class boundaries. The classifier achieved an accuracy of 98.59%, indicating that the latent space is highly discriminative.

Further analysis using dimensionality reduction techniques such as PCA shows clear clustering and separation between classes in the latent space. These results suggest that the autoencoder successfully captured class-relevant structure, despite being trained without labels. This demonstrates the effectiveness of the encoder as a feature extractor.

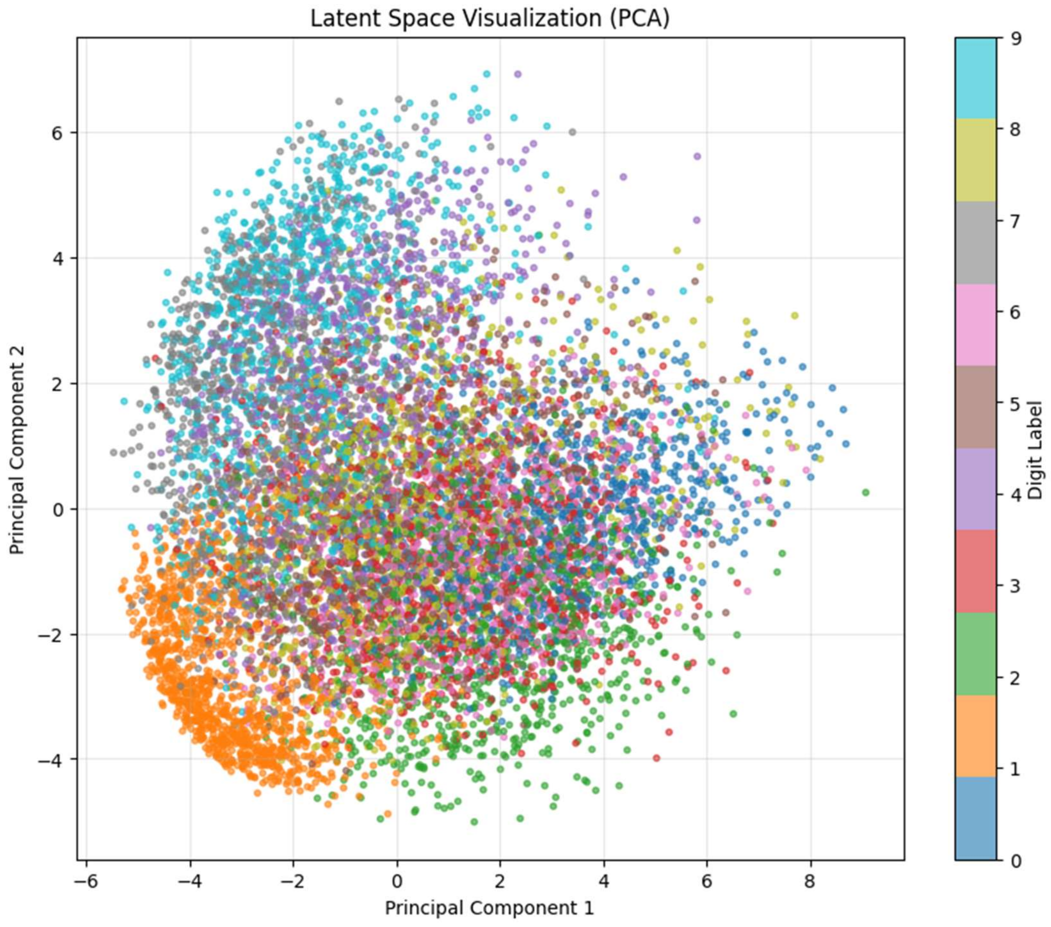
- SVM Classification Accuracy: 98.59%
- Confusion matrix:



- Classification Report:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	980
1	0.99	1.00	0.99	1135
2	0.99	0.98	0.98	1032
3	0.98	0.99	0.98	1010
4	0.99	0.99	0.99	982
5	0.98	0.98	0.98	892
6	0.99	0.99	0.99	958
7	0.98	0.98	0.98	1028
8	0.99	0.98	0.98	974
9	0.98	0.98	0.98	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

- Visualized latent space



7.0. Comparison with TensorFlow/Keras:

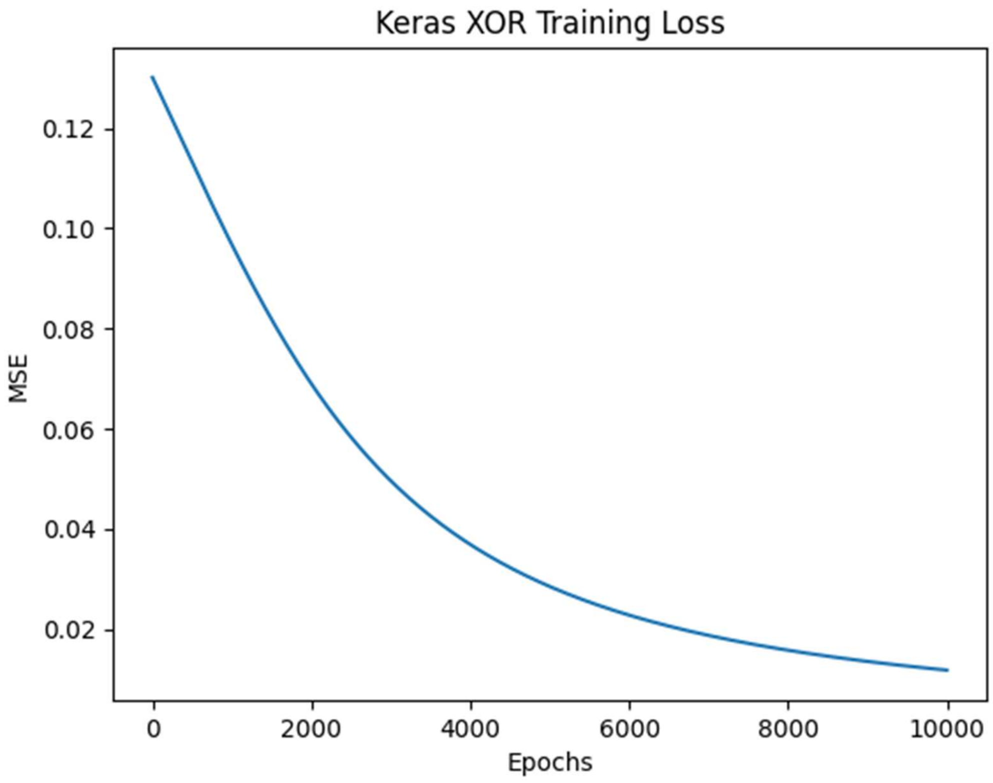
A comparative analysis was conducted between the custom neural network library and TensorFlow/Keras implementations of the same models. For the XOR problem, both implementations achieve similar convergence and final accuracy, though TensorFlow trains significantly faster due to hardware optimization and highly optimized numerical kernels.

For the autoencoder, TensorFlow again demonstrates faster training and slightly more stable convergence. However, the qualitative reconstruction results are similar in both cases. This similarity validates the correctness of the custom implementation while highlighting the trade-off between performance and transparency.

TensorFlow provides speed, scalability, and abstraction, making it suitable for production and large-scale experiments. In contrast, the custom library offers full visibility into each computational step, making it highly valuable for learning and experimentation. Both approaches achieve comparable outcomes, but serve different purposes

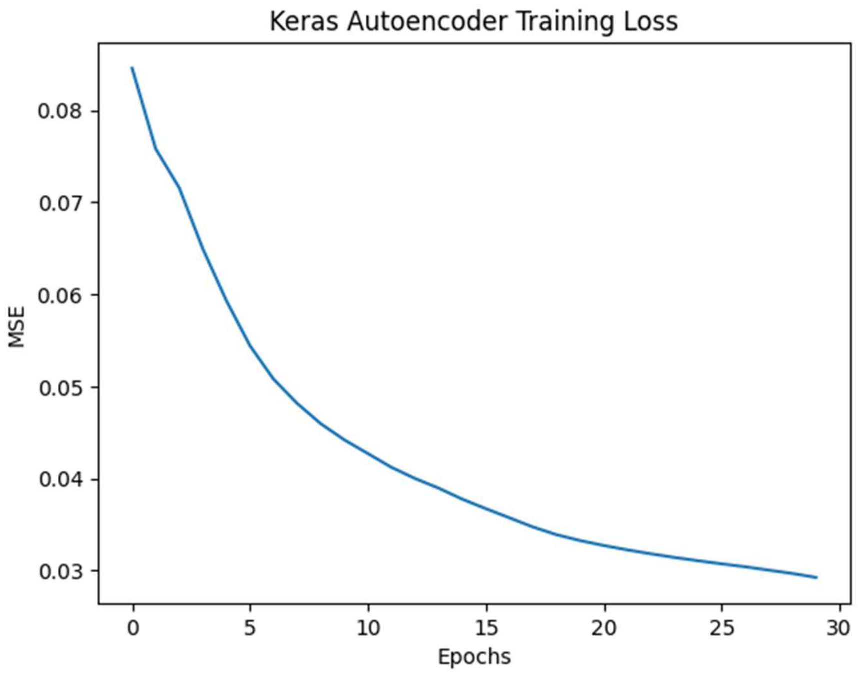
.

- XOR loss:

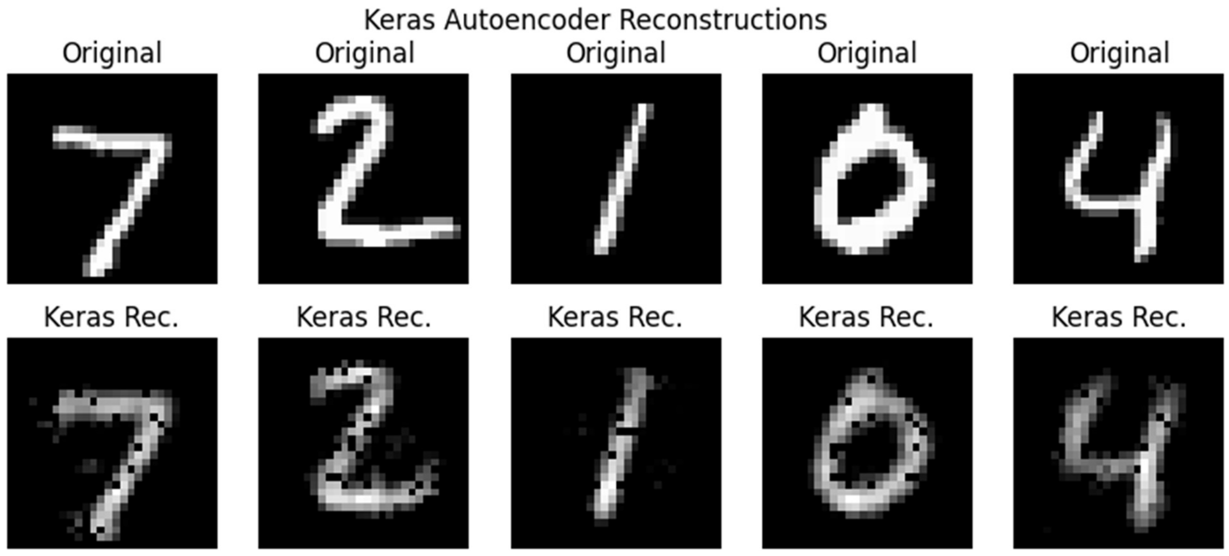


Training Time: 442.4413 seconds
Final Loss: 0.011759

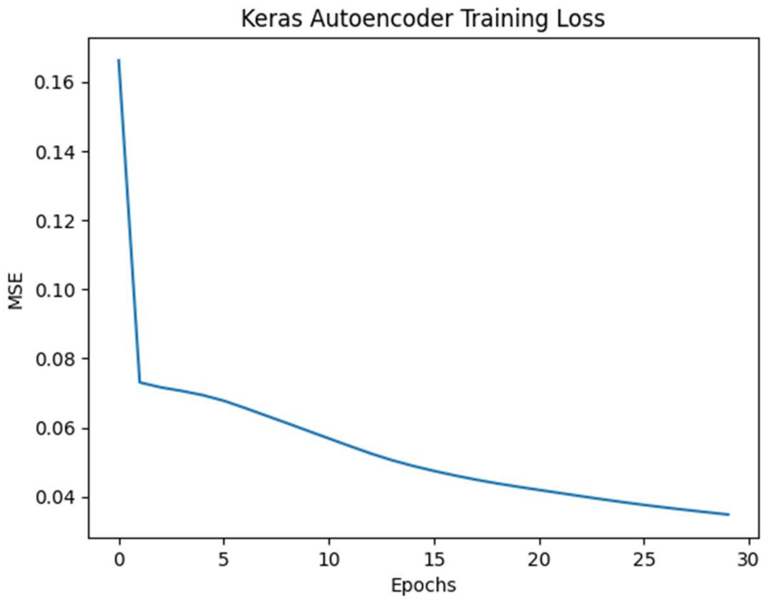
- Autoencoder with relu:



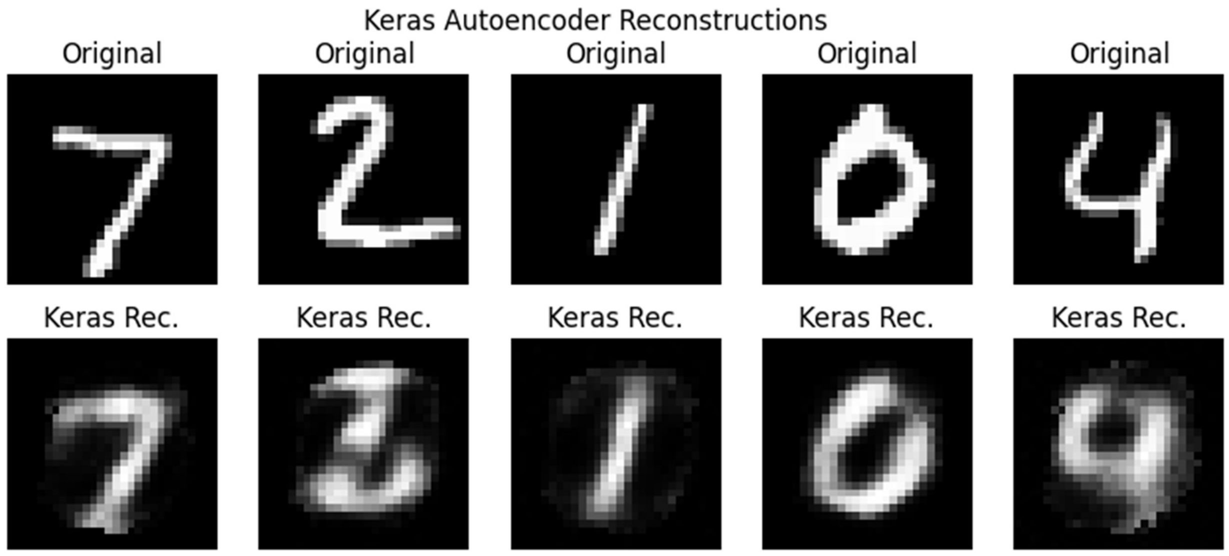
Training Time: 300.76 seconds
Final Train Loss: 0.029243
Final Val Loss: 0.028720



- Autoencoder with sigmoid:



Training Time: 279.84 seconds
Final Train Loss: 0.029243
Final Val Loss: 0.028720



8.0. Challenges Faced and Lessons Learned:

One of the main challenges encountered during the project was weight initialization. Initially, simple scaling of weights worked well for the XOR problem but failed for the autoencoder, leading to poor convergence and black output images. This issue was resolved by adopting Xavier initialization, which balances the variance of activations across layers and improves training stability.

Another challenge involved training dynamics. Early versions of the library did not properly support batch-based training, which negatively affected the autoencoder's learning behavior. After correcting the batching mechanism and tuning the learning rate, the reconstruction quality improved significantly. These issues highlighted how implementation details that may not affect small problems like XOR can become critical for deeper or larger networks.

Overall, the project reinforced the importance of proper initialization, batching, and hyperparameter selection. It also demonstrated that building neural networks from scratch provides valuable insight into training behavior that is often hidden in high-level frameworks.

9.0 Conclusion:

In this project, a neural network library built from scratch was successfully extended and validated on multiple tasks, including XOR classification, autoencoder-based representation learning, and downstream SVM classification. The results demonstrate that the library is correct, flexible, and capable of learning meaningful representations.

The comparison with TensorFlow/Keras shows that while optimized frameworks offer superior performance, a custom implementation provides unmatched transparency and educational value. Through this project, we gained a deeper understanding of neural network training, representation learning, and the practical challenges involved in implementing machine learning systems from the ground up.

10.0. GitHub link:

https://github.com/mmaarrwa/neural-network-library