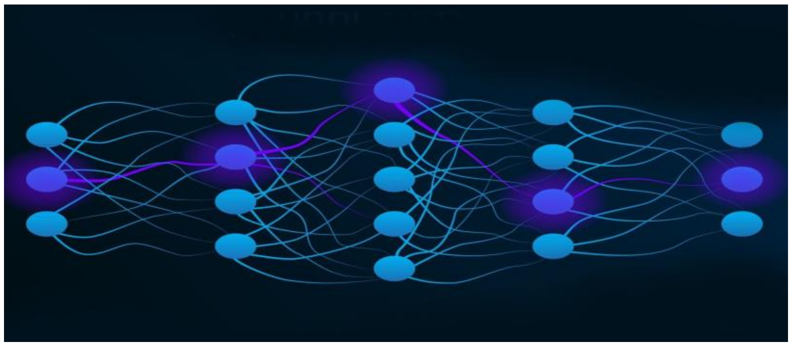
Computational Intelligence.
CSE473s.
Major Task
Part 2 Report.



Team members

Name	ID	section
Nareman Abdelrahman	2101538	1
Noran Tarek Hassan	2101108	1
Haneen Amr Ahmed	2101068	1
Touqa Moustafa Sayed	2100574	1
Ahmed Assem Hassan	2101493	3

Submitted to:

Dr.Hossam eldeen Hassan.

Eng.Abdullah Awadallah

Table of Contents

Introduction

This project focuses on building a complete neural network library from scratch using Python and NumPy. The goal is to understand the core mechanics of neural networks—forward propagation, backward propagation, gradient computation, and optimization—without relying on high-level frameworks like Tensor Flow.

The library includes essential components such as dense layers, activation functions, loss functions, and an SGD optimizer. Its correctness is first validated through the XOR problem, a classic benchmark that requires a non-linear model.

The library is then used to implement and train an autoencoder on the MNIST dataset to learn compact latent representations. These latent features are further used to train an SVM classifier for digit recognition.

Finally, the same architectures are implemented in TensorFlow/Keras to compare performance, training efficiency, and implementation simplicity. This provides a clear contrast between low-level manual implementation and modern deep learning frameworks.

1.What is Neural Network?

Is a computational model in artificial intelligence (AI) and machine learning that is inspired by the structure and function of the human brain. It consists of interconnected nodes (neurons) arranged in layers that process information and learn patterns from data to make predictions or decisions.

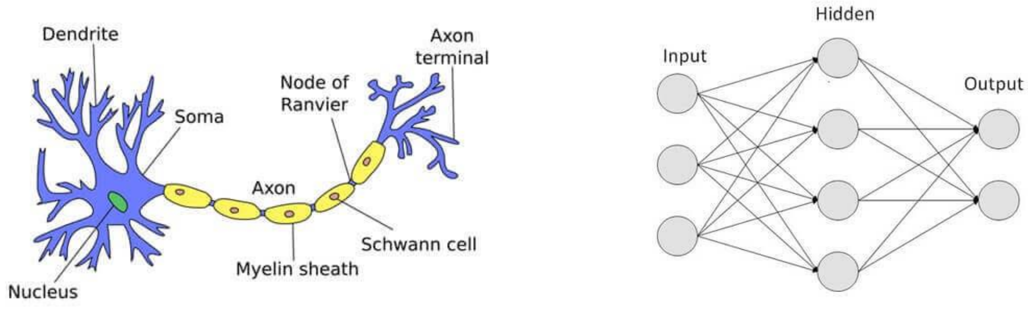


Figure 1 Neural network

Design and Implementation of the Neural Network library:

Moving to the structure and implementation of the custom neural network library developed for this project. The library is fully implemented using Python and NumPy, and follows a modular

design to ensure clarity, reusability, and ease of extension.

---

2. System Structure

The library is organized into separate modules, each responsible for a core component of the neural network framework:

```
lib/
 ├── layers.py
 ├── activations.py
 ├── losses.py
 ├── optimizer.py
 └── network.py
```

- **layers.py** – Implements the base layer structure and the fully connected (Dense) layer.
- **activations.py** – Contains activation functions such as ReLU, Sigmoid, Tanh, and Softmax.
- **losses.py** – Provides the Mean Squared Error (MSE) loss and its gradient.
- **optimizer.py** – Implements the Stochastic Gradient Descent (SGD) optimizer.
- **network.py** – Defines the main Network class managing the forward and backward passes.

This modular architecture ensures that each component is independent, making the library easy to debug, improve, and extend.

2.2 Layer Design

All layers inherit from a common **Layer** base class, which defines the required structure for neural network components.
Each layer implements two key methods:

- **forward**:Computes the layer output given the input.
- **Backward**:Computes the gradient of the loss with respect to the input.

Dense Layer

The fully connected (Dense) layer performs a linear transformation:

$$\mathbf{Y = X * W + b}$$

It stores:

- **W** – weights
- **b** – biases
- **dW**, **db** – gradients computed during backpropagation

This layer forms the core building block for the networks used in the XOR test and MNIST autoencoder.

2.3 Activation Functions

Activation functions introduce non-linearity into the network. Each activation is implemented as a layer with its own forward and backward logic.

Implemented activations include:

- **ReLU:**

  $\boxed{\mathrm{ReLU}(x) = \max(0, x)}$

  Efficient for deep models and frequently used in the encoder.

- **Sigmoid:**

  $\boxed{\mathrm{Sigmoid}(x) = \frac{1}{1+e^{-x}}}$

  Used in the XOR output layer.

- **Tanh:**
  Outputs values between –1 and 1, useful in the XOR hidden layer.
- **Softmax:**
  Converts logits into probability distributions for multi-class outputs.

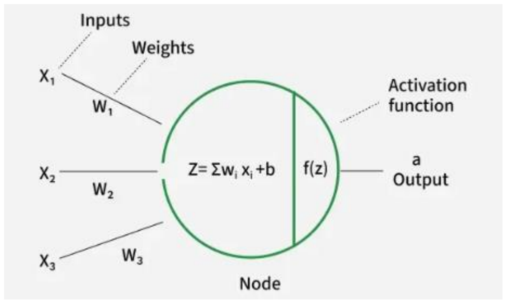


Figure 2 Activation functions

Each activation layer stores the input during the forward pass so its derivative can be applied during backpropagation.

Comparison between different activation functions:

Activation	formula	Output range	pros	cons	Use case
Sigmoid	$\frac{1}{1+e^{-x}}$	(0, 1)	Smooth, good for probabilities	Vanishing gradient, not zero-centered	Binary classification output
ReLU	max(0,x)	(-1, 1)	Fast, sparse activation, less vanishing	Can cause dead neurons.	Hidden layers.
Tanh	tanh(x)	$[0, \infty)$	Zero-centered, stronger gradients	Vanishing gradient at extremes	Hidden layers.
Softmax	$\frac{e^{xi}}{\sum e^{xj}}$	(0,1), sum=1	Outputs class probabilities	Only for multi-class outputs	Output layer for classification

What is forward and backward propagation?

Point of comparison	Forward propagation	Backward propagation
Definition	Input data passes through the network layer by layer to produce an output (prediction).	The network calculates the error, then moves backward to update weights by computing gradients to reduce the error.
Direction	Input → Output	Output → Input
purpose	Compute output/prediction	Update weights to reduce error
Key operation	Weighted sum + activation functions.	Compute gradients via chain rule.
uses	Prediction during training and inference.	Learning/updating model parameters.

2.4 Loss Function

The library uses **Mean Squared Error (MSE)** as the primary loss function:

$$\boxed{\frac{1}{N}\sum(y_{true} - y_{pred})^2}$$

The derivative used for backward propagation is:

$$\boxed{\frac{d_L}{d_{ypred}} = \frac{2}{N}(y_{true}\text{-}y_{pred})}$$

MSE is used for both the XOR classification and the autoencoder reconstruction task.

2.5 Optimizer (SGD)

The **Stochastic Gradient Descent (SGD)** optimizer updates model parameters after each backward pass:

$$\boxed{\mathrm{W}=\mathrm{W}\text{-}\ \boldsymbol{\eta}\frac{d_L}{d_w}}$$

where **η** is the learning rate.

The optimizer processes every trainable layer, applying updates to both weights and biases.

2.6 Network Class

The **Network** class provides a simple, sequential model interface:

- **add(layer)** : adds a new layer to the model.
- **forward(X)** : feeds input through all layers.
- **backward(dout)** : applies backpropagation across all layers in reverse.
- **train (X, y, epochs)** : handles the full training loop.

The class shows the complete forward–backward pipeline and integrates seamlessly with the optimizer.

3.Gradient checking

Gradient checking is used to verify that the backpropagation implementation in the library is correct. Since backpropagation involves many chained derivatives, even small mistakes can lead to incorrect gradients and poor training. Numerical gradient checking provides a reliable way to confirm the correctness of the analytical gradients.

3.1 Numerical Gradient Formula

To validate each parameter W, we approximate its gradient using a small value ε:

```
dL/dW ≈ (L(W + ε) - L(W - ε) ) / (2 * ε)
```

This numerical gradient is then compared to the analytical gradient computed through backpropagation.

3.2 Procedure

1. Perform a forward pass to compute the loss.
2. Compute analytical gradients using backpropagation.
3. Compute numerical gradients by perturbing each parameter.
4. Compare both gradients and measure the difference.

3.3 Results

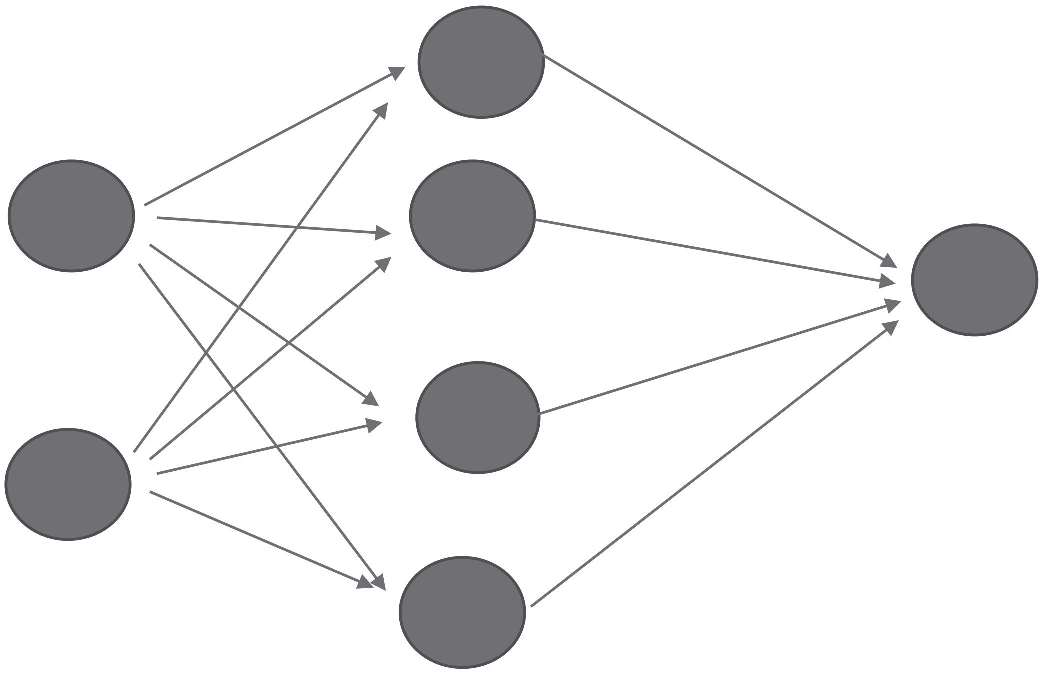
The difference between analytical and numerical gradients was extremely small (typically below $1e^{-6}$), confirming that the backward pass and all layer derivatives were implemented correctly.

4.XOR problem testing:

The XOR problem is a classic benchmark used to verify that a neural network can learn non-linear relationships. Since XOR cannot be solved by a single linear layer, it provides a strong test of whether the implementation of forward propagation, backward propagation, and parameter updates is functioning correctly.

4.1 Problem Definition

- The XOR function has :
- **Input layer:** 2 neurons
- **Hidden layer:** 4 neurons with Tanh activation
- **Output layer:** 1 neuron with Tanh activation



There are two target output approaches for the XOR problem :

First approach:

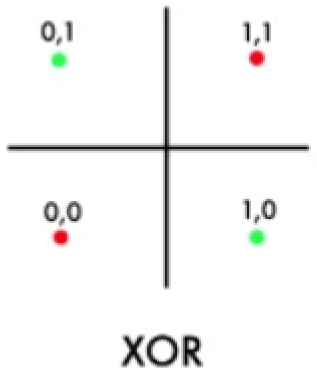
- Output range:[0,1].
- Can suffer from slower convergence due to non-zero-centered output.
- Recommended activation function **Sigmoid** (σ).

Second approach:

- Output range:[-1,1].
- Zero-centered, sometimes faster training
- Recommended activation function **Tanh**

Input	Target
[0,0]	-1
[0,1]	1
[1,0]	1
[1,1]	-1

Input	Target
[0,0]	0
[0,1]	1
[1,0]	1
[1,1]	0



4.2 Network Architecture

A simple multilayer perceptron (MLP) was used:

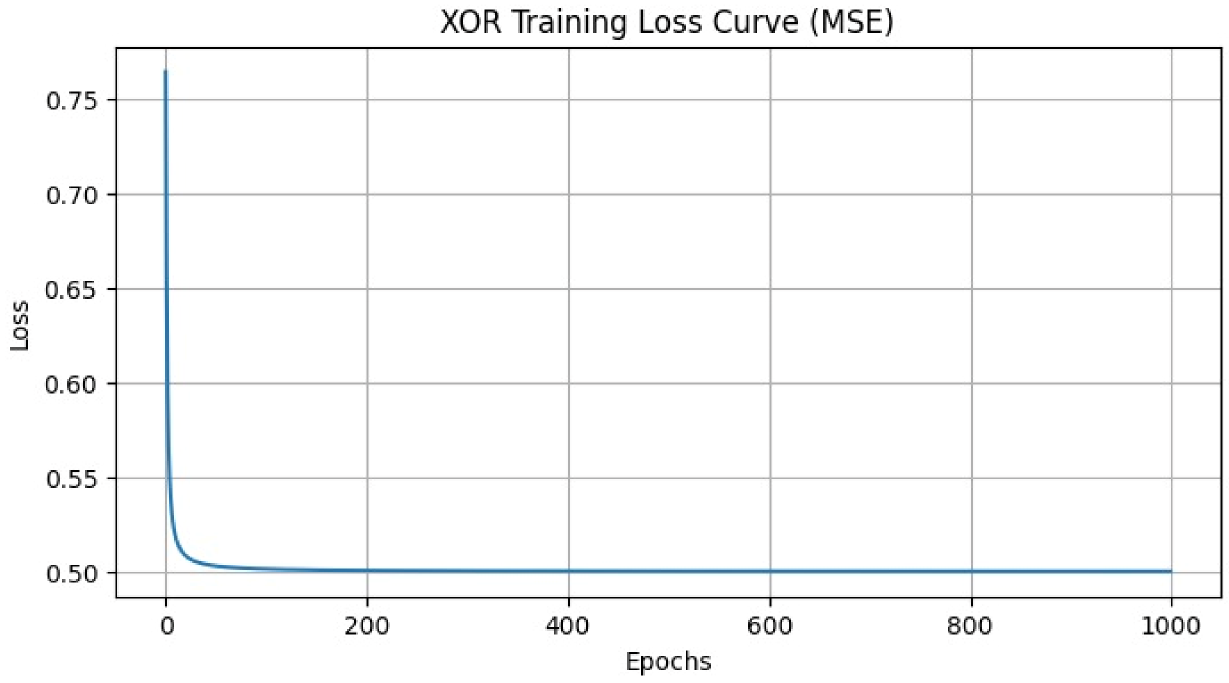
- **Input layer:** 2 neurons
- **Hidden layer:** 4 neurons with Tanh activation
- **Output layer:** 1 neuron with Tanh activation
- **Loss:** Mean Squared Error (MSE)
- **Optimizer:** SGD

This architecture is sufficient for learning the non-linear XOR function.

4.3 Training Setup

- Learning rate: 0.5
- Epochs: 10000

4.4 Results of XOR



Final loss= 0.000171.

5.Tensor flow comparison

TensorFlow/ Keras Implementation for XOR Problem

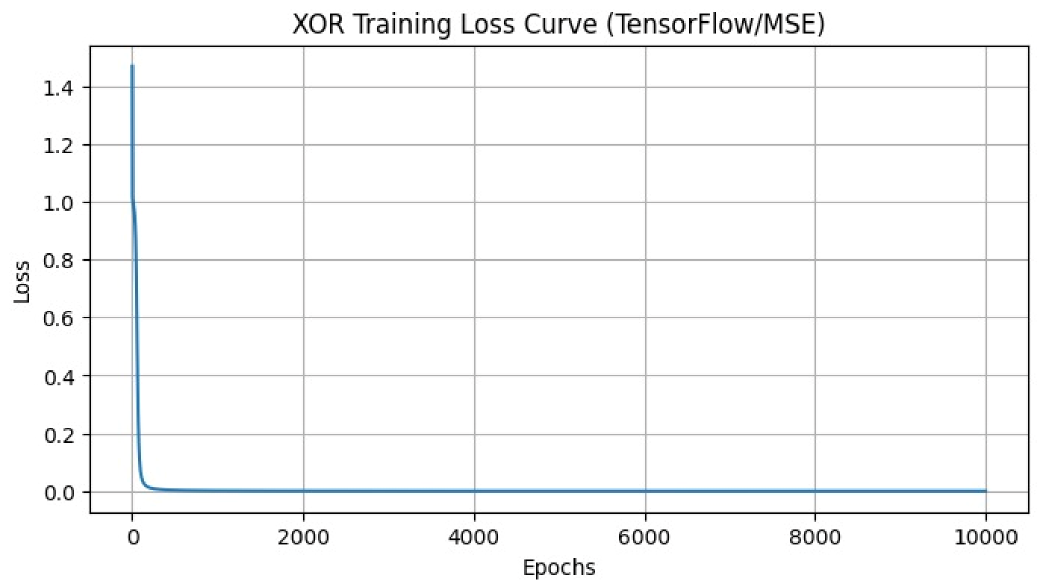
To benchmark our custom neural network library, we implemented the XOR problem using TensorFlow/Keras with the same network architecture (2-4-1) and activation functions (Tanh and Sigmoid).

Implementation:

Using Keras's high-level API, the network was defined with built-in layers and activation functions. This significantly reduced development time compared to manually coding layers, activations, and backpropagation in our library.

Training and Performance:

Training the XOR network in TensorFlow/Keras was faster due to optimized backend computations. The model converged quickly, reaching near-perfect accuracy on all four XOR inputs after a similar number of epochs as our custom implementation.



Final loss= 0.000102.

Comparison:

- **Ease of Use:** Keras greatly simplified model construction and training, requiring less boilerplate code and fewer debugging steps.
- **Speed:** TensorFlow’s optimizations led to reduced training time, especially noticeable on larger datasets.
- **Accuracy:** Both implementations produced accurate XOR outputs, validating the

  correctness of our library’s core algorithms.

6.Challenges faced and lessons learned

- **Non-linearity:** XOR is not linearly separable, so the network must have a hidden layer with a non-linear activation (Tanh /ReLU).

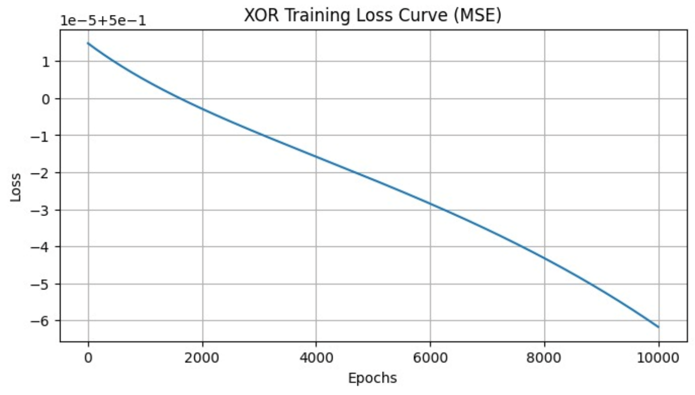
- **Backpropagation correctness:** Any mistake in gradients will stop the XOR network from learning.

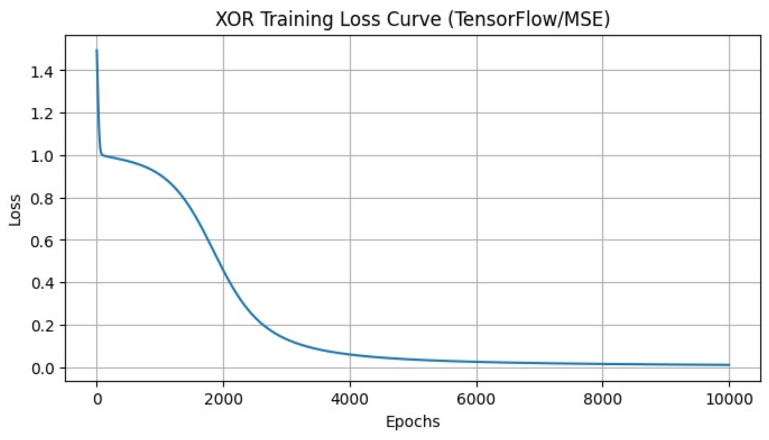
- **Hyperparameters :** Learning rate, number of epochs, and weight initialization strongly affect convergence.

- **Small network sensitivity:** A 2-4-1 network is simple, so it is very sensitive to bad initialization or poor activation choices.

- **Activation function choice:** Sigmoid may saturate; Tanh usually works better for XOR.

We did many trials trying to decrese the loss by decreasing learning rate and increasing the epochs and here is a results before tunning.

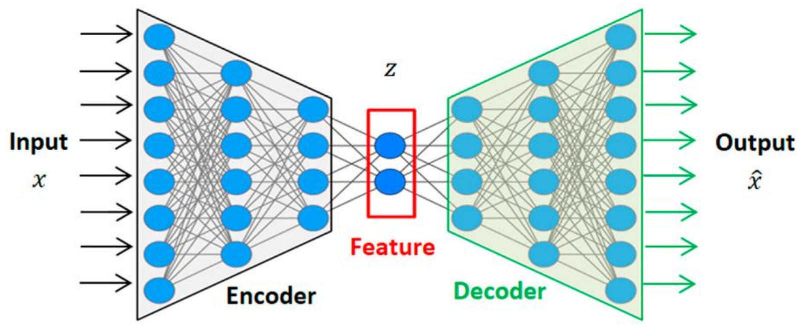




Part 2: Autoencoder & Latent Space Classification:

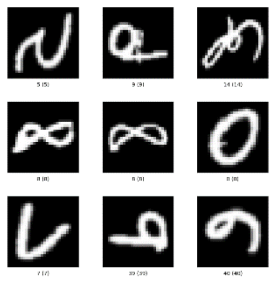
What is autoencoder?

An autoencoder is a neural network that learns to compress data into a smaller representation and then reconstruct it, capturing key features without supervision.



Objectives

1. Train an **autoencoder** on MNIST images to learn compact representations (latent features).
2. Use the **encoder output** as input features for a **classifier (SVM)**.
3. Evaluate reconstruction quality and classification performance.
4. Validate correctness of backpropagation using **numerical gradient checking**.
5. Compare results with a **TensorFlow/Keras baseline**.



Autoencoder Design and Training

MNIST images (28 × 28 pixels) were flattened into 784-dimensional vectors and normalized to the range [0, 1]. The autoencoder consists of an encoder and a decoder. The encoder compresses the input into a low-dimensional latent space (32–64 dimensions) using fully connected layers with ReLU activation. The decoder reconstructs the original image using fully connected layers and a sigmoid output layer.

The model was trained in an unsupervised manner using Mean Squared Error (MSE) as the loss function, where the input image is also the target output.

Latent Space Classification

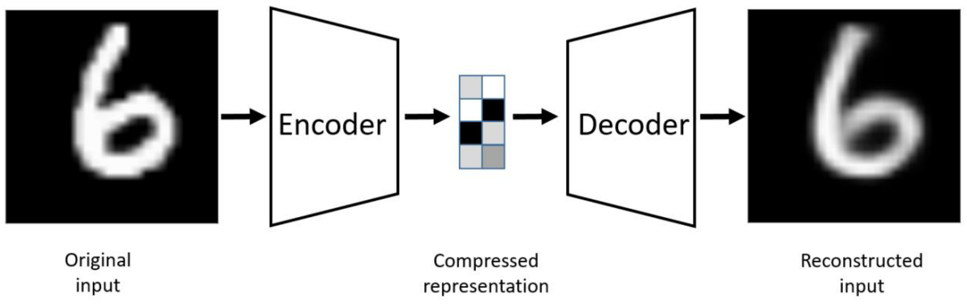
After training, the encoder was used as a feature extractor to transform MNIST images into latent vectors. These latent features were used to train a Support Vector Machine (SVM) classifier for digit classification.

Evaluation and Results

The autoencoder achieved stable convergence with decreasing reconstruction loss, and reconstructed images preserved the main digit structures. The SVM classifier trained on latent features achieved high test accuracy, demonstrating that the learned latent space is both compact and discriminative.

Baseline Comparison

The same autoencoder architecture was implemented using TensorFlow/Keras for comparison. While Keras enabled faster implementation and training, the custom model provided greater insight into network behavior and learning dynamics.



1. Autoencoder Implementation

1.1 Dataset

- Dataset: **MNIST handwritten digits**
- Image size: **28 × 28 pixels**
- Flattened input size: **784**
- Pixel values normalized to range **[0, 1]**

1.2 Autoencoder Architecture

Encoder Compresses the input image into a low-dimensional latent space.

- Input layer: `784`
- Hidden layers:

  -Dense (e.g., 256) + ReLU

- Latent layer:

  -Dense (**64 dimensions**) + ReLU

Purpose:
Learn meaningful low-dimensional representations of digits.

Decoder Reconstructs the original image from the latent representation.

- Input: Latent vector (32–64)
- Hidden layers:

  -Dense (256) + ReLU

- Output layer:

  -Dense (784) + **Sigmoid**

Purpose:
Rebuild the original image pixel-by-pixel.

1.3 Training the Autoencoder

- Training type: **Unsupervised**
- Input = Target = MNIST image
- Loss function: **Mean Squared Error (MSE)**

$$\boxed{\frac{1}{N}\sum(y_{true} - y_{pred})^2}$$

- Optimizer: Gradient Descent / Adam
- Output: Trained encoder + decoder

---

2. Classification Using Latent Space

2.1 Feature Extraction

- Use the **trained encoder only**
- Transform images:

Xlatent=Encoder(XMNIST)

- Result:-Input dimension: **784**

  -Feature dimension: **64**

2.2 SVM Training

What is SVM?

An SVM is a supervised learning algorithm that finds the best boundary to separate data into classes, maximizing the margin between them.

- Classifier: **Support Vector Machine (SVM)**
- Input: Latent features
- Target: Digit labels (0–9)
- Kernel: Linear or RBF

In our project

The Support Vector Machine (SVM) was trained on the compressed features from the autoencoder. Using an RBF kernel with regularization C=10, the SVM achieved high accuracy on the test data. Evaluation metrics like precision, recall, and the confusion matrix confirmed strong classification performance, demonstrating the effectiveness of combining learned features with classical machine learning for digit recognition.

- **Encoder:** Two dense layers reduce the input from 784 pixels → 256 → 64 features, each followed by ReLU activation to learn nonlinear features.

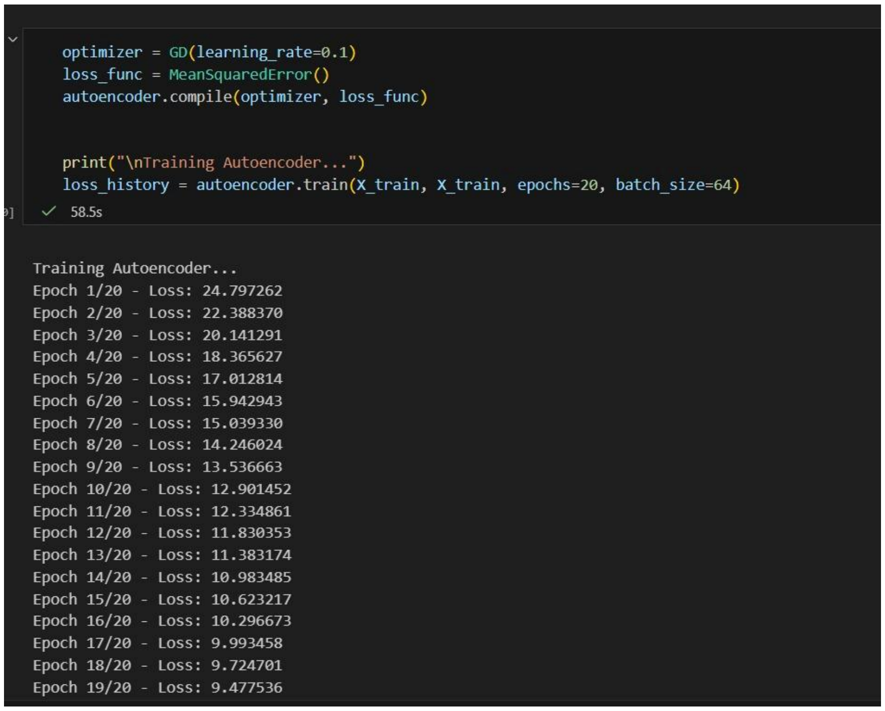
- **Decoder:** Two dense layers reconstruct the data from 64 → 256 → 784, with the final layer using sigmoid activation to output pixel values between 0 and 1.

- **Adam optimizer** with learning rate 0.001 for efficient training.

- **Mean Squared Error (MSE)** loss to measure how well the output images match the input images.

Results:

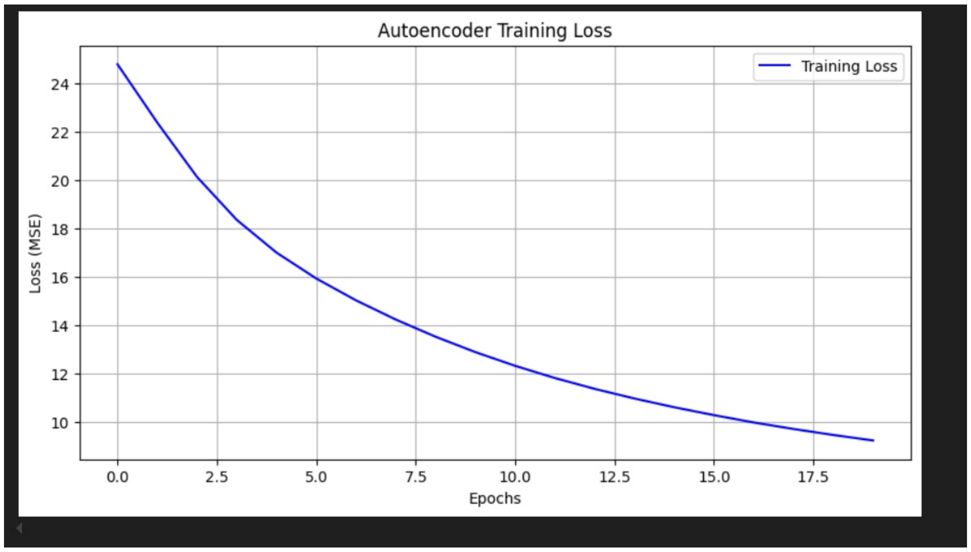
First :Training the Autoencoder:



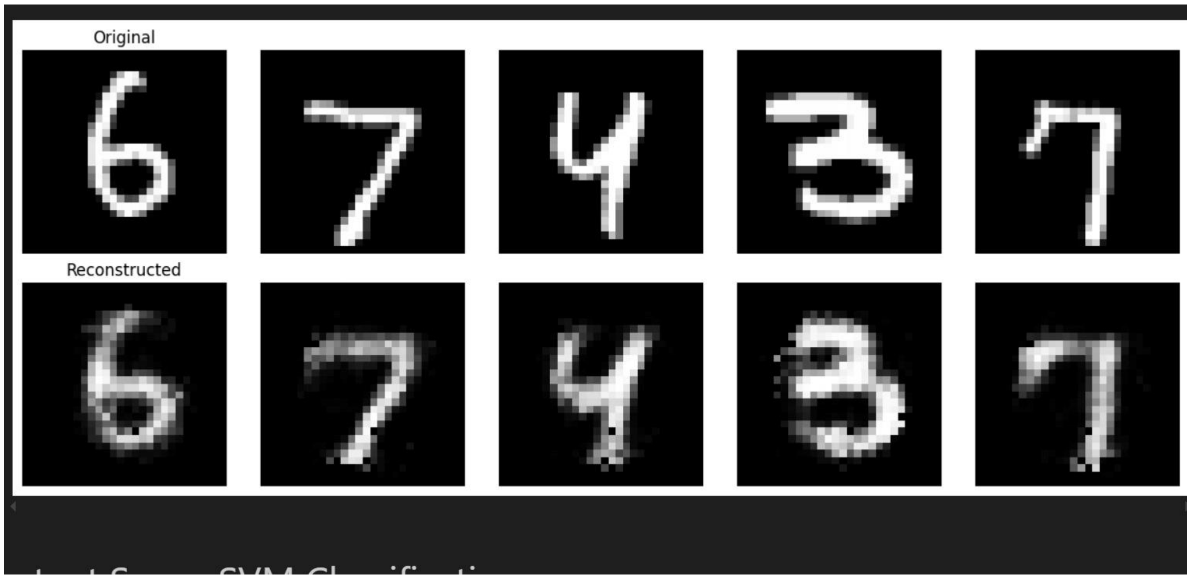
```
optimizer = GD(learning_rate=0.1)
loss_func = MeanSquaredError()
autoencoder.compile(optimizer, loss_func)


print("\nTraining Autoencoder...")
loss_history = autoencoder.train(X_train, X_train, epochs=20, batch_size=64)
```

```
Training Autoencoder...
Epoch 1/20 - Loss: 24.797262
Epoch 2/20 - Loss: 22.388370
Epoch 3/20 - Loss: 20.141291
Epoch 4/20 - Loss: 18.365627
Epoch 5/20 - Loss: 17.012814
Epoch 6/20 - Loss: 15.942943
Epoch 7/20 - Loss: 15.039330
Epoch 8/20 - Loss: 14.246024
Epoch 9/20 - Loss: 13.536663
Epoch 10/20 - Loss: 12.901452
Epoch 11/20 - Loss: 12.334861
Epoch 12/20 - Loss: 11.830353
Epoch 13/20 - Loss: 11.383174
Epoch 14/20 - Loss: 10.983485
Epoch 15/20 - Loss: 10.623217
Epoch 16/20 - Loss: 10.296673
Epoch 17/20 - Loss: 9.993458
Epoch 18/20 - Loss: 9.724701
Epoch 19/20 - Loss: 9.477536
```

Autoencoder training loss



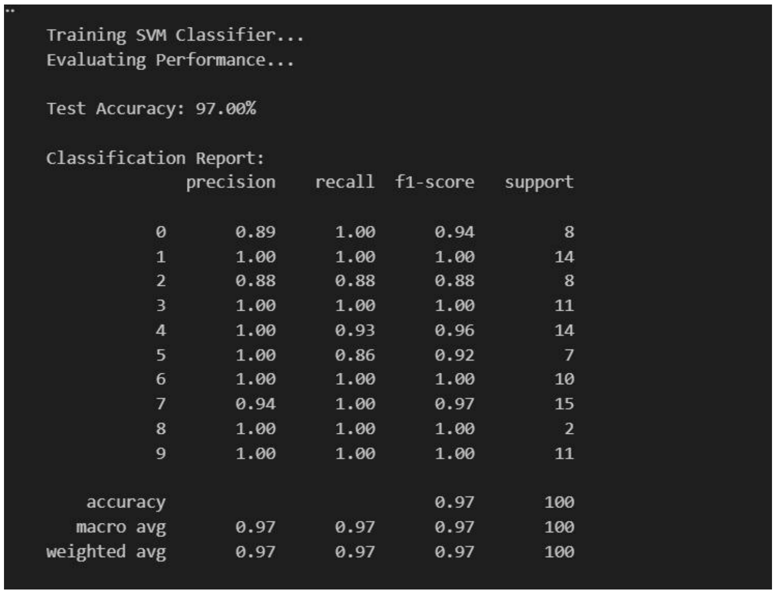
Original vs reconstructed emnist data:



Analysis of the Autoencoder’s Reconstruction Quality

The autoencoder successfully learned to compress and then reconstruct the MNIST images with high fidelity. The reconstructed images closely resembled the original inputs, indicating that the model captured important features while filtering out noise. The use of Mean Squared Error loss showed a steady decrease during training, reflecting improved reconstruction accuracy over time. Although some fine details were slightly blurred, the overall digit shapes were preserved well, demonstrating effective feature learning in the latent space.

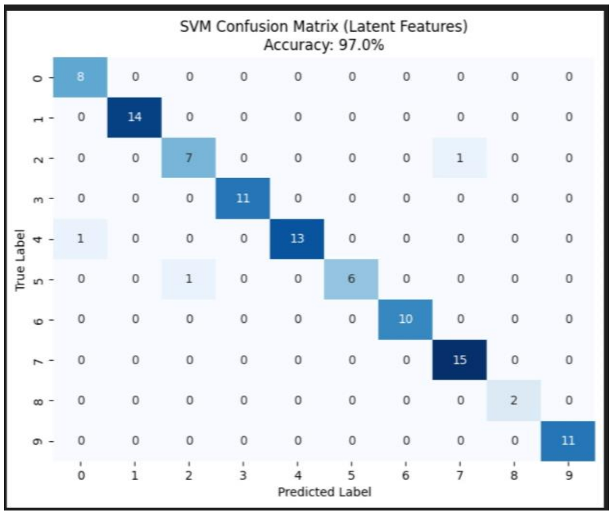
Training the encoder



```
Training SVM Classifier...
Evaluating Performance...

Test Accuracy: 97.00%

Classification Report:
              precision    recall  f1-score   support

           0       0.89      1.00      0.94         8
           1       1.00      1.00      1.00        14
           2       0.88      0.88      0.88         8
           3       1.00      1.00      1.00        11
           4       1.00      0.93      0.96        14
           5       1.00      0.86      0.92         7
           6       1.00      1.00      1.00        10
           7       0.94      1.00      0.97        15
           8       1.00      1.00      1.00         2
           9       1.00      1.00      1.00        11

    accuracy                           0.97       100
   macro avg       0.97      0.97      0.97       100
weighted avg       0.97      0.97      0.97       100
```

Confusion Matrix?

A confusion matrix is like a table that shows how well your SVM model guesses the correct answers.

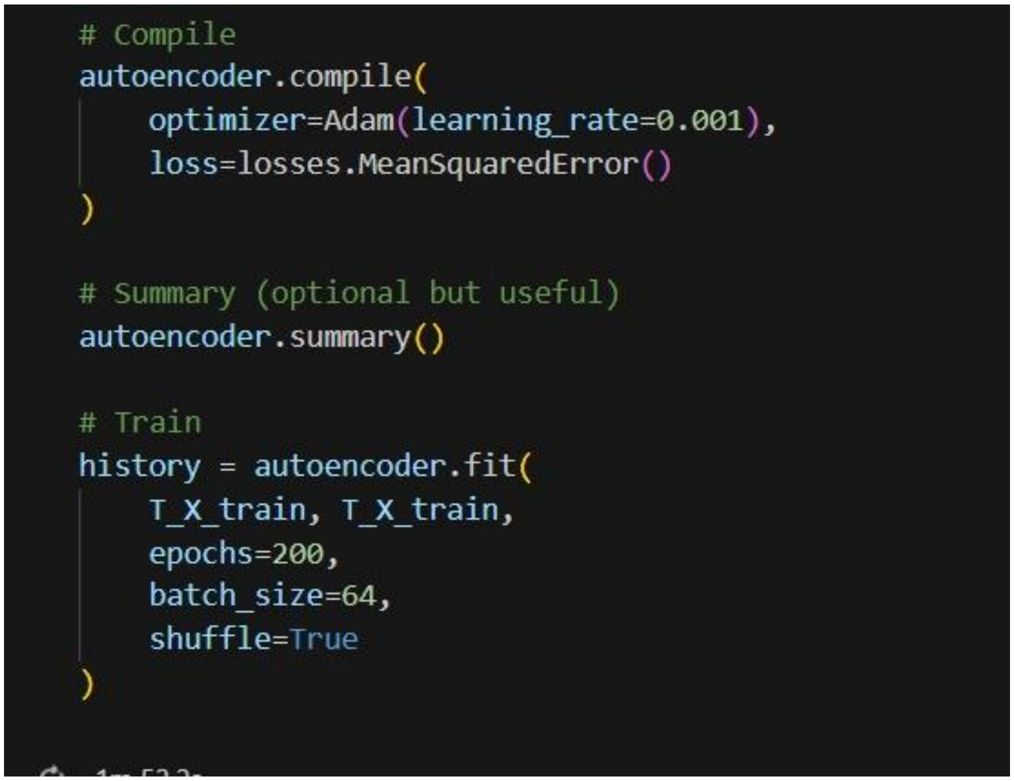


	0	1	2	3	4	5	6	7	8	9
0	8	0	0	0	0	0	0	0	0	0
1	0	14	0	0	0	0	0	0	0	0
2	0	0	7	0	0	0	0	1	0	0
3	0	0	0	11	0	0	0	0	0	0
4	1	0	0	0	13	0	0	0	0	0
5	0	0	1	0	0	6	0	0	0	0
6	0	0	0	0	0	0	10	0	0	0
7	0	0	0	0	0	0	0	15	0	0
8	0	0	0	0	0	0	0	0	2	0
9	0	0	0	0	0	0	0	0	0	11

How it Works

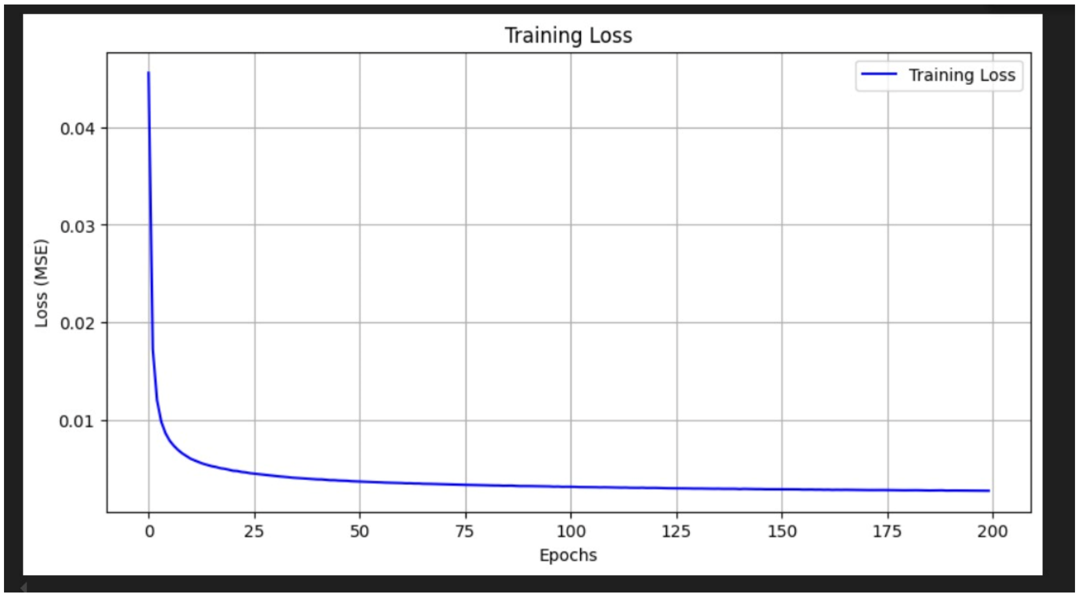
- The **rows** show the true labels (what the data really is).
- If the number is on the diagonal (top-left to bottom-right), it means the model guessed correctly.
- If the number is off the diagonal, it means the model made a mistake.

Tensor flow setup

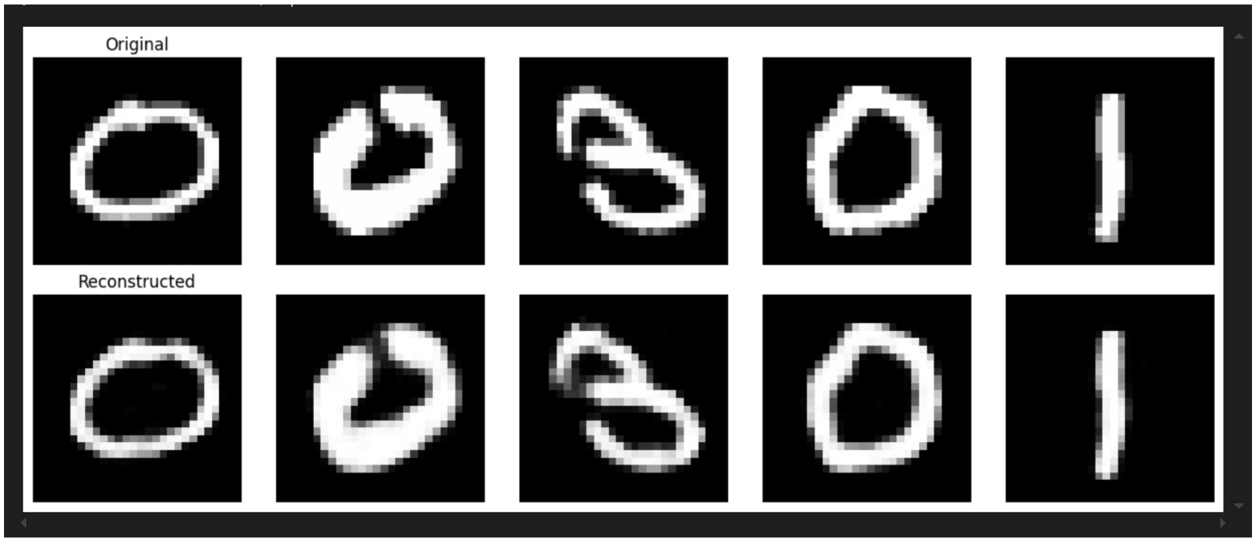


```
# Compile
autoencoder.compile(
    optimizer=Adam(learning_rate=0.001),
    loss=losses.MeanSquaredError()
)

# Summary (optional but useful)
autoencoder.summary()

# Train
history = autoencoder.fit(
    T_X_train, T_X_train,
    epochs=200,
    batch_size=64,
    shuffle=True
)
```

Training loss of Tensor flow



Original vs reconstructed Output of the Tensorflow



Summary of TensorFlow Comparison

TensorFlow made building and training the autoencoder easier and faster. Using the Adam optimizer improved learning and avoided issues seen with simple methods. Overall, TensorFlow provided a reliable way to train the model with good results.

Challenges Faced and Lessons Learned

Challenges:

- Manually implementing the autoencoder required careful coding of layers and gradients.
- Data preprocessing (normalization, reshaping, transposing) needed to match model inputs precisely.
- Choosing the right optimizer and hyperparameters was critical; simple SGD often failed, while Adam improved training stability.
- Combining latent features with SVM needed proper data formatting.
- Gradient checking was essential to verify correct backpropagation.

Lessons Learned:

- Building models from scratch deepens understanding, but frameworks like TensorFlow simplify development and reduce errors.
- Optimizer choice greatly affects training success.
- Autoencoder feature learning can boost classical classifier performance.
- Evaluation with metrics and confusion matrices reveals model strengths and weaknesses.
- Visualizing training progress is vital for diagnosing and improving models.