Faculty of Engineering

Ain Shams University

Neural Network Library & Advanced Applications

Team#6

NAME	ID
AHMED MOSTAFA ABDOU	2101004
AYMAN ABD AL-ELAH AL SHEBAH	2101760
AHMED SAYED MOHAMED	2101798

Submitted To:

- Dr. Hossam Hassan
- Eng. Abdallah Awdallah

Table of Contents

Neural Network Library & Advanced Applications 1

❖ Introduction: 4

- ➢ Project Overview 4
- ➢ Scope and Objectives 4

1. Library Design and Architecture Choices 6

1.1 Modular Architecture 6

1.2 Forward and Backward Propagation Flow 7

2. Results from the XOR Test 7

2.1 Experimental Setup 7

2.2 Training Results 8

2.3 Final Predictions 8

2.4 Conclusion of Validation 8

3 MNIST Autoencoder Baseline 9

3.1 Dataset Description 9

3.2 Autoencoder Architecture 9

4 Autoencoder fabrication 9

4.1 Encoder Design 10

4.2 Decoder Design 11

4.3 Autoencoder Integration 11

4.4 Forward Propagation Logic 12

4.5 Design Flexibility and Modularity 12

5. Autoencoder Training 13

5.1 Dataset Loading and Integrity Verification 13

5.2 Data Preprocessing 13

5.3 Training Configuration 14

5.4 Training Loop and Optimization 14

5.5 Training Time Measurement 15

5.6 Loss Monitoring and Visualization 15

5.7 Training results 15
6. Autoencoder testing 16
6.1 Test Sample Selection 16
6.2 Forward Inference on Test Data 16
6.3 Reconstruction Visualization 17
6.4 Qualitative Analysis 17
6.5 Results 17
7.1 Latent Feature Extraction 18
7.2 SVM Model Configuration 18
7.3 Training Procedure 19
7.4 Classification Performance Evaluation 19
7.4.1 Accuracy 19
7.4.2 Classification Report 19
7.4.3 Confusion Matrix Analysis 20
8. Keras Baseline Implementation and Comparison 21
8.1 Reproducibility and Experimental Setup 21
8.2 XOR Classification Baseline (Keras) 21
8.2.1 Problem Definition 21
8.2.2 Model Architecture 21
8.2.3 Training Configuration 22
8.2.4 Results 22
8.3 MNIST Autoencoder Baseline (Keras) 22
8.3.1 Dataset and Preprocessing 22
8.3.2 Autoencoder Architecture 22
8.3.3 Training Configuration 23
9. Comparison 24

❖ Introduction:

➢ Project Overview

- In the domain of Computational Intelligence, understanding the mathematical underpinnings of deep learning is as critical as mastering high-level frameworks. This project focuses on the development of a foundational Neural Network library built entirely from scratch using only Python and NumPy.
- The primary objective is to move beyond the "black box" abstraction provided by modern libraries like TensorFlow. By manually implementing core components: including dense layers, activation functions, loss calculations, and optimization algorithms, we gain a rigorous understanding of the backpropagation algorithm and the flow of gradients through a computational graph.

➢ Scope and Objectives

- The project is structured into three main phases, designed to bridge the gap between theoretical concepts and practical application:

    1. **Library Implementation & Validation:** The initial phase involves constructing a modular library capable of defining sequential models. This includes implementing forward and backward passes for fully connected layers and non-linear activations (Sigmoid, Tanh, Softmax, ReLU). The library's correctness is rigorously validated by solving the **XOR problem**, a classic benchmark for testing non-linear classification capabilities.
    2. **Unsupervised Learning (Autoencoder):** Leveraging the custom library, we implement an autoencoder architecture designed for image reconstruction on the MNIST dataset. This phase tests the library's ability to handle higher-dimensional data and optimization landscapes more complex than simple logic gates.

3. **Transfer Learning & Classification:** The final phase explores feature extraction. By utilizing the trained encoder's latent space representations, we train a Support Vector Machine (SVM) to classify handwritten digits. This demonstrates how neural networks can serve as powerful feature extractors for traditional machine learning algorithms.

- Finally, the project concludes with a critical comparative analysis, benchmarking the custom implementation against industry-standard frameworks to evaluate performance, ease of use, and computational efficiency.

1. Library Design and Architecture Choices

- This section details the architectural decisions made during the development of the custom Neural Network library. The library follows a modular Object-Oriented Programming (OOP) design, prioritizing flexibility, readability, and ease of debugging.

1.1 Modular Architecture

- The library is structured into distinct modules, each handling a specific aspect of the neural network pipeline. This separation of concerns allows for easy extension (e.g., adding new layer types or optimizers) without modifying the core logic.

  - **Layer Abstraction (Layer.py):** A base abstract class `layer` defines the contract for all network components. It enforces the implementation of two critical methods:
    - `forward(x)`: Computes the output given the input and caches necessary data.
    - `Backword (grad_output)`: Computes the gradient with respect to the input and weights.
  - **Dense Layer (Layers.py):** The core fully connected layer implementation. It manages its own weight (W) and biases (b).
    - *Architecture Choice:* We utilized **Xavier/Glorot uniform initialization** for weights to maintain variance across layers, preventing vanishing or exploding gradients during the initial passes.
  - **Activation Functions (Activation.py):** Activations like `Tansh` and `sigmoid` are implemented as subclasses of `layer`. This treats activations as just another step in the computational graph, simplifying the forward/backward flow.
  - **Network Management (Network.py):** The `Network` class acts as the orchestrator. It maintains a list of layers and manages the training loop. It decouples the optimization logic from the model definition, allowing users to "compile" the model with different loss functions and optimizers.

1.2 Forward and Backward Propagation Flow

The data flow follows a strict pipeline managed by the `Network` class:

1. **Forward Pass:** Input data flows sequentially through the list of layers. The output of Layer (N) becomes the input of Layer (N+1).
2. **Loss Calculation:** The final output is compared against the true labels using the `MeanSquaredError` class.
3. **Backward Pass:** The gradient of the loss flows in reverse. Each layer's `backward()` method receives the gradient from the subsequent layer, computes its local gradients (stored in `self.dW`, `self.db`), and passes the gradient w.r.t input to the previous layer.
4. **Parameter Update:** The `SGD` optimizer iterates through all layers and updates parameters using the stored gradients.

2. Results from the XOR Test

- The XOR (Exclusive OR) problem is a classic benchmark for neural networks because it is not linearly separable. A single perceptron cannot solve it; it requires a multi-layer architecture with non-linear activation functions.

2.1 Experimental Setup

- To validate the library, we constructed a Multi-Layer Perceptron (MLP) with the following architecture:

  - **Input Layer:** 2 Neurons (representing binary inputs 0/1).
  - **Hidden Layer:** 4 Neurons with **Tanh** activation.
  - **Output Layer:** 1 Neuron with **Sigmoid** activation (outputs probability between 0 and 1).
  - **Optimizer:** Stochastic Gradient Descent (SGD) with Learning Rate: $\eta = 0.5$.
  - **Loss Function:** Mean Squared Error (MSE).
  - **Training Duration:** 10,000 Epochs.

2.2 Training Results

- The network was trained on the standard XOR truth table: X = [[0,0], [0,1], [1,0], [1,1]], Y_{true} = [[0], [1], [1], [0]]
- **Convergence:** The model successfully converged. The Mean Squared Error started high (approx. 0.104) due to random initialization and decreased steadily, stabilizing near 0.00103 by epoch 10,000.

2.3 Final Predictions

- After training, the model was evaluated on the input set. The raw predicted probabilities demonstrate that the network has successfully learned the XOR logic.

Input (A, B)	True Label	Predicted Probability	Rounded Output
(0, 0)	0	0.0228	0
(0, 1)	1	0.9527	1
(1, 0)	1	0.9505	1
(1, 1)	0	0.0556	0

2.4 Conclusion of Validation

- The results confirm that the library's backpropagation algorithm is correctly implemented. The network successfully learned non-linear decision boundaries, proving the correctness of the `Dense` layer gradients, `Tanh`/`Sigmoid` derivatives, and the `SGD` update rule.

3 MNIST Autoencoder Baseline

3.1 Dataset Description

The MNIST dataset consists of grayscale handwritten digit images of size **28 × 28 pixels**, flattened into 784-dimensional vectors. Pixel values are normalized to the range **[0, 1]**, which is appropriate for sigmoid output activations.

Labels are not used, as the task is unsupervised.

3.2 Autoencoder Architecture

The autoencoder is composed of two main components:

- **Encoder:**
    - Fully connected layer reducing the input from 784 dimensions to a **latent space of 64 dimensions** using ReLU activation.
- **Decoder:**
    - Fully connected layer reconstructing the original 784-dimensional input using a **sigmoid activation**, matching the normalized pixel range.

This shallow autoencoder architecture is intentionally simple to align with the custom implementation and emphasize core learning behavior rather than architectural complexity.

4 Autoencoder fabrication

This section describes the design and implementation of a **fully custom autoencoder architecture** built using the developed neural network framework. The autoencoder is constructed by explicitly defining the **Encoder**, **Decoder**, and **Autoencoder** classes, ensuring complete control over layer operations, activation functions, and training behavior.

Unlike high-level libraries such as Keras, this implementation relies entirely on manually defined layers, loss functions, and optimizers, providing transparency into the learning process and enabling fine-grained experimentation.

4.1 Encoder Design

The **Encoder** is responsible for compressing the input image into a low-dimensional latent representation.

The input consists of grayscale images of shape **(28, 28)**. Since fully connected layers operate on vectors, the input is first flattened into a one-dimensional representation.

Encoder Architecture:

- **Flatten:** Converts the input from (28 × 28) to a 784-dimensional vector
- **Dense (784 → 256):** First feature extraction layer
- **ReLU activation:** Introduces non-linearity and improves gradient flow
- **Dense (256 → latent_dim):** Compresses features into the latent space
- **Tanh activation:** Bounds latent values and stabilizes representation

This architecture progressively reduces dimensionality while preserving essential image features.

The encoder can be expressed as:

$$\mathbf{z} = \tanh\left(W_2 \cdot \text{ReLU}(W_1 \cdot \mathbf{x} + b_1) + b_2\right)$$

where $\mathbf{z}$is the latent vector.

4.2 Decoder Design

The **Decoder** reconstructs the original image from the latent representation generated by the encoder.

Its structure mirrors the encoder, expanding the compressed representation back to the original image dimensions.

Decoder Architecture:

- **Dense (latent_dim → 256):** Expands latent features
- **ReLU activation:** Restores non-linear structure
- **Dense (256 → 784):** Produces flattened image output
- **Sigmoid activation:** Constrains pixel values to the range [0, 1]
- **Reshape:** Converts the flattened output back to (28 × 28) image format

The sigmoid activation is particularly important, as it matches the normalized pixel range used during training.

4.3 Autoencoder Integration

The **Autoencoder** class combines the encoder and decoder into a single end-to-end network.

A key design decision in this implementation is **layer flattening**. The encoder and decoder layers are concatenated into a single layer list:

$$\text{layers} = \text{Encoder.layers} + \text{Decoder.layers}$$

This ensures seamless compatibility with the existing **SGD optimizer and backpropagation engine**, which iterates sequentially over network layers.

4.4 Forward Propagation Logic

The forward pass is explicitly defined for clarity and modularity:

1. The input image is passed through the encoder to generate the latent vector
2. The latent vector is then passed through the decoder
3. The output is the reconstructed image

$$\hat{\mathbf{x}} = \text{Decoder}(\text{Encoder}(\mathbf{x}))$$

Backward propagation is handled automatically by the parent Network class, which propagates gradients through the combined layer list in reverse order.

4.5 Design Flexibility and Modularity

The autoencoder design allows flexible experimentation through:

- Configurable **latent dimensionality**
- Customizable **activation functions** for both encoder and decoder
- Easy extension to deeper architectures without modifying the training pipeline

This modular structure enables direct comparison with equivalent Keras models while maintaining full control over learning dynamics.

5. Autoencoder Training

This section describes the dataset preparation, integrity verification, preprocessing steps, and the training procedure used to optimize the fabricated autoencoder model.

5.1 Dataset Loading and Integrity Verification

The MNIST handwritten digit dataset is used to train and evaluate the autoencoder. It consists of:

- **60,000 training images**
- **10,000 test images**
- Image resolution: **28 × 28 grayscale**

To ensure a **strict separation between training and testing data**, an explicit duplicate check is performed. Each training image is converted into a byte representation and stored in a hash set, enabling constant-time lookup. All test images are then compared against this set.

This verification confirms that:

- The training and test sets are **fully disjoint**
- No data leakage occurs during evaluation

Such validation is critical for ensuring unbiased performance assessment and reproducibility.

5.2 Data Preprocessing

Prior to training, the image data undergoes the following preprocessing steps:

- Conversion to **32-bit floating-point format**
- Normalization of pixel values to the range **[0, 1]**

$$x_{\text{norm}} = \frac{x}{255}$$

Normalization aligns with the decoder's sigmoid output activation, ensuring numerical stability and meaningful reconstruction loss values.

5.3 Training Configuration

The autoencoder is trained using the following settings:

- **Training type:** Unsupervised learning
- **Input = Target:** The network learns to reconstruct its input
- **Loss function:** Mean Squared Error (MSE)
- **Optimizer:** Stochastic Gradient Descent (SGD)
- **Epochs:** 500
- **Batch size:** Configurable (defined externally)

The number of training steps per epoch is computed as:

$$\text{steps per epoch} = \frac{N_{\text{train}}}{\text{batch size}}$$

This ensures uniform coverage of the dataset during each epoch.

5.4 Training Loop and Optimization

Training is performed using a **manual epoch–batch loop**, providing full visibility into the optimization process.

For each batch:

1. A mini-batch is sampled from the training data
2. The autoencoder performs a forward pass
3. The reconstruction loss is computed using MSE
4. Gradients are propagated backward through the network
5. Parameters are updated using SGD

The optimizer's step function encapsulates this sequence, directly interacting with the custom Network class.

The average loss per epoch is computed and stored for post-training analysis.

5.5 Training Time Measurement

Training duration is explicitly measured to evaluate computational efficiency:

$$\text{Training Time} = t_{\text{end}} - t_{\text{start}}$$

This metric is useful for comparing performance against optimized frameworks such as Keras and for assessing the scalability of the custom implementation.

5.6 Loss Monitoring and Visualization

To track learning progress, the average reconstruction loss is recorded at the end of each epoch. A loss curve is then plotted, showing:

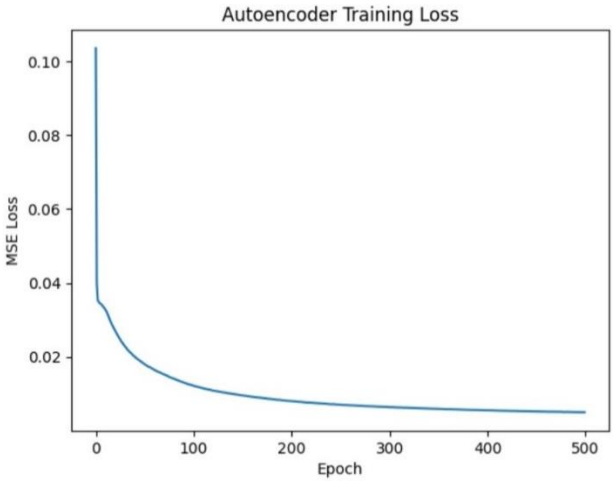
- Rapid initial loss reduction
- Gradual convergence over later epochs

This behavior indicates successful learning of compact latent representations and stable optimization dynamics.

5.7 Training results

```
✓ 52m 18.0s

Training Data: (60000, 28, 28)
Test Data: (10000, 28, 28)
Hashing training data for fast lookup...
Checking all test images...
Verified: The test set is completely unique (disjoint) from the training set.
Training Data Shape: (60000, 28, 28)
Testing Data Shape: (10000, 28, 28)
Model compiled with Latent Dim: 64
Starting Training...
Epoch 1/500 | Avg Loss: 0.103628
Epoch 2/500 | Avg Loss: 0.039847
Epoch 3/500 | Avg Loss: 0.035268
Epoch 4/500 | Avg Loss: 0.034898
Epoch 5/500 | Avg Loss: 0.034526
Epoch 6/500 | Avg Loss: 0.034329
Epoch 7/500 | Avg Loss: 0.034081
Epoch 8/500 | Avg Loss: 0.033771
Epoch 9/500 | Avg Loss: 0.033405
Epoch 10/500 | Avg Loss: 0.033101
Epoch 11/500 | Avg Loss: 0.032671
Epoch 12/500 | Avg Loss: 0.032192
Epoch 13/500 | Avg Loss: 0.031602
Epoch 14/500 | Avg Loss: 0.030900
Epoch 15/500 | Avg Loss: 0.030237
Epoch 16/500 | Avg Loss: 0.029570
...
Epoch 498/500 | Avg Loss: 0.004943
Epoch 499/500 | Avg Loss: 0.004951
Epoch 500/500 | Avg Loss: 0.004957
Training completed in 3121.76 seconds.
```



6. Autoencoder testing

This section evaluates the trained autoencoder using unseen test data. The goal is to assess the model's reconstruction capability and generalization performance through qualitative visual analysis.

6.1 Test Sample Selection

To ensure an unbiased evaluation, a subset of **10 images** is randomly selected from the MNIST test dataset. Sampling is performed **without replacement** to avoid duplicate images within the evaluation set.

Random selection ensures that the evaluation reflects the model's performance on diverse and previously unseen digit patterns rather than on carefully curated examples.

6.2 Forward Inference on Test Data

The selected test samples are passed through the trained autoencoder using a direct forward pass:

$$\hat{\mathbf{x}} = \text{Autoencoder}(\mathbf{x})$$

Internally, the model:

1. Encodes the input image into a latent representation
2. Decodes the latent vector back into the image space

This explicit forward operation highlights the modular separation between the encoder and decoder while validating their seamless integration.

6.3 Reconstruction Visualization

To qualitatively evaluate reconstruction performance, original test images are displayed alongside their reconstructed counterparts.

- **Top row:** Original MNIST images
- **Bottom row:** Corresponding reconstructed images

This side-by-side comparison allows direct visual inspection of:

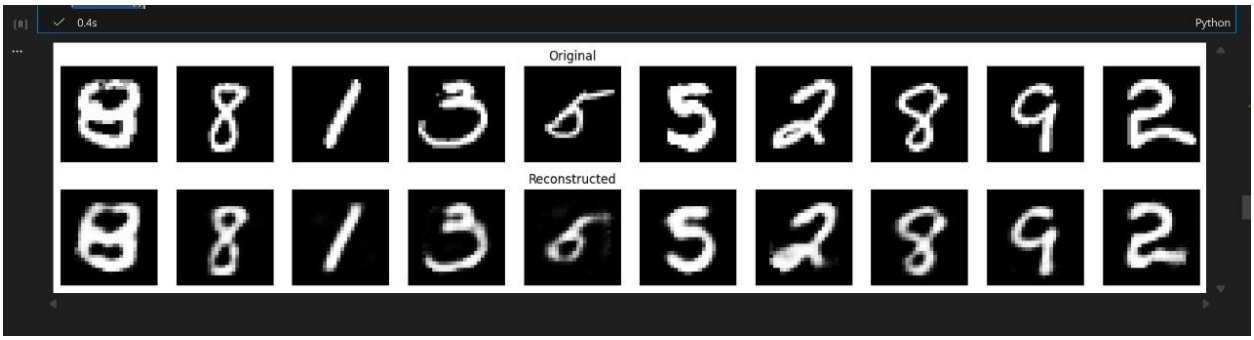
- Structural fidelity of reconstructed digits
- Preservation of digit shape and stroke continuity
- Loss of fine details due to dimensionality reduction

6.4 Qualitative Analysis

The reconstructed images closely resemble the originals, indicating that the autoencoder has successfully learned a compact representation of the data. Minor smoothing and blurring effects are observed, which are expected outcomes of compression through a low-dimensional latent space.

Despite these limitations, the reconstructions retain the essential semantic content of each digit, demonstrating effective generalization to unseen data.

6.5 Results



7. SVM Classification Using Latent Features

This section evaluates the quality of the learned latent representations by employing a **Support Vector Machine (SVM)** classifier. Instead of training a deep classifier end-to-end, the encoder of the trained autoencoder is used as a **feature extractor**, and a classical machine learning model is trained on the compressed features.

7.1 Latent Feature Extraction

After training the autoencoder, the **encoder component** is isolated and used to transform the MNIST images into a compact latent representation.

- Input shape: **(N, 28, 28)**
- Output shape: **(N, latent_dim)**

Since the encoder uses a **Tanh activation** at its output layer, the resulting latent features are bounded within the range **[−1, 1]**, which improves numerical stability and benefits kernel-based classifiers such as SVMs.

Both training and test datasets are passed through the encoder using the custom framework's forward propagation mechanism, ensuring consistent feature extraction without relying on external libraries.

7.2 SVM Model Configuration

A **Support Vector Classifier (SVC)** with a **Radial Basis Function (RBF)** kernel is used to perform digit classification in the latent space.

SVM Hyperparameters:

- **Kernel:** RBF (Gaussian)
- **C:** 10
- **Gamma:** Scale
- **Random state:** 42

The RBF kernel is selected due to the expected non-linear separability of the latent features. A moderate regularization parameter (C = 10) balances margin maximization and classification accuracy, which is particularly effective when working with compact and denoised representations.

7.3 Training Procedure

The SVM is trained using:

- **Latent training features** extracted from the encoder
- **Original MNIST digit labels (0–9)**

This decoupled training strategy allows the classifier to focus solely on decision boundaries while relying on the autoencoder for representation learning.

7.4 Classification Performance Evaluation

The trained SVM is evaluated on the latent features extracted from the test dataset using multiple performance metrics.

7.4.1 Accuracy

Overall classification accuracy is computed as:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Predictions}}$$

The achieved accuracy demonstrates that the latent space preserves class-discriminative information despite being learned in an unsupervised manner.

7.4.2 Classification Report

A detailed classification report is generated, including:

- **Precision**
- **Recall**
- **F1-score**

for each digit class. These metrics provide insight into per-class performance and highlight any class-specific weaknesses.

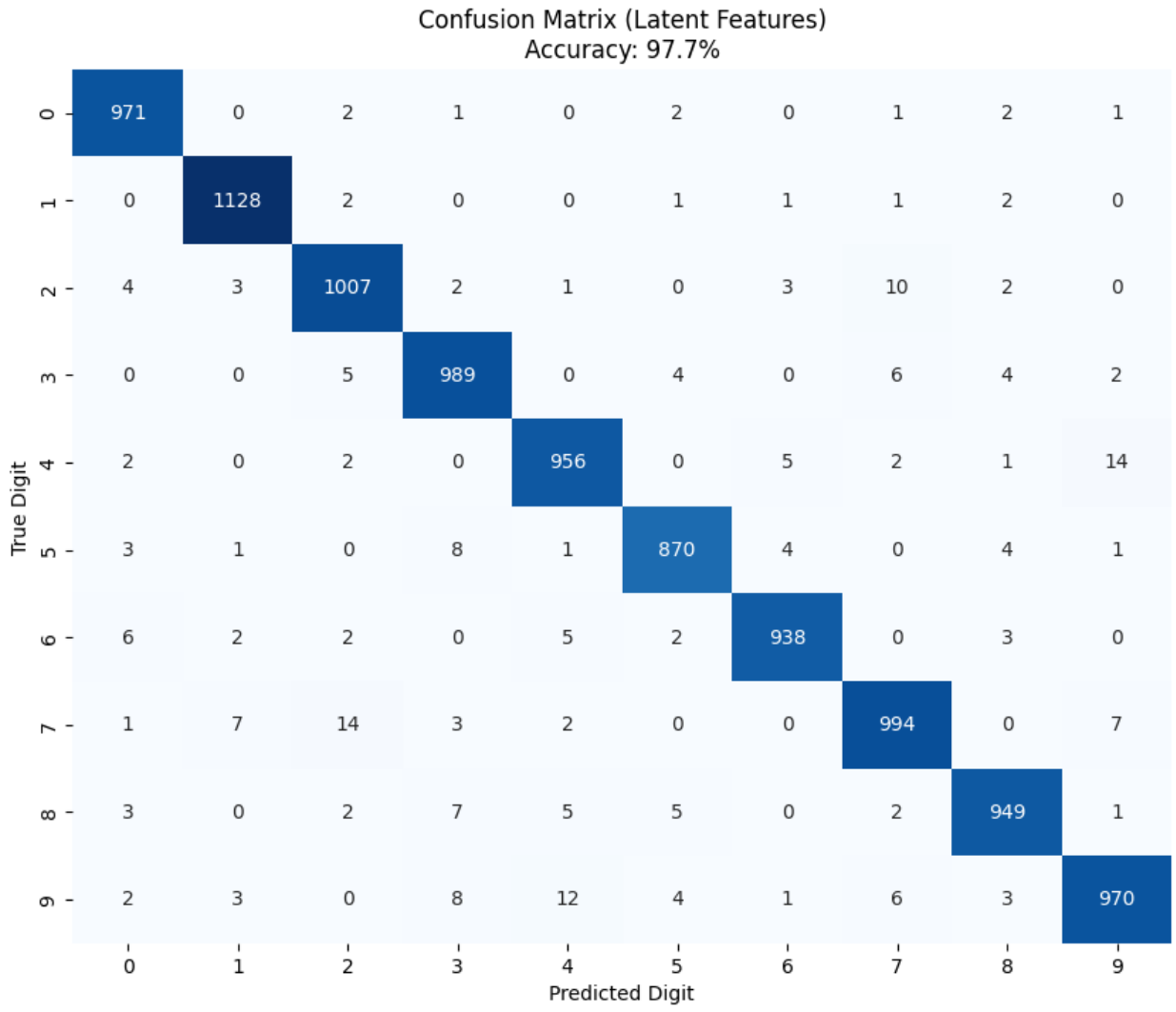
7.4.3 Confusion Matrix Analysis

A confusion matrix is visualized using a heatmap to analyze misclassification patterns across digit classes.

Key observations include:

- Strong diagonal dominance, indicating high correct classification rates
- Confusions primarily between visually similar digits (e.g., 4 and 9, 3 and 5)

This analysis confirms that errors are largely semantic rather than random.



8. Keras Baseline Implementation and Comparison

This section presents baseline implementations using **TensorFlow Keras** to validate and benchmark the behavior of the custom neural network framework. Two representative tasks are considered: a **nonlinear classification problem (XOR)** and an **unsupervised reconstruction task (MNIST autoencoder)**.
The architectures, loss functions, and optimization settings are selected to closely match those used in the custom implementation, enabling a fair and meaningful comparison.

8.1 Reproducibility and Experimental Setup

To ensure deterministic behavior across multiple runs, random seeds are fixed for both **NumPy** and **TensorFlow**. This guarantees consistent weight initialization, data shuffling, and training dynamics, which is essential when comparing convergence speed and final performance against a custom-built framework.

8.2 XOR Classification Baseline (Keras)

8.2.1 Problem Definition

The XOR problem is a classical benchmark used to verify a neural network's ability to model **nonlinearly separable functions**. It consists of four two-dimensional input samples with binary targets.

8.2.2 Model Architecture

The Keras XOR model follows the same structure used in the custom implementation:

- **Input:** 2 features
- **Hidden layer:** 4 neurons with **tanh activation**
- **Output layer:** 1 neuron with **sigmoid activation**

This architecture ensures that differences in performance arise from implementation details rather than architectural choices.

8.2.3 Training Configuration

- **Optimizer:** Stochastic Gradient Descent (SGD)
- **Learning rate:** 0.3
- **Loss function:** Mean Squared Error (MSE)
- **Batch size:** 1
- **Epochs:** 5000

Training time is explicitly measured to assess computational efficiency.

8.2.4 Results

The Keras model successfully learns the XOR mapping, achieving a low final loss and producing correct binary predictions. This confirms correct nonlinear learning behavior and serves as a reference point for validating the custom XOR implementation.

8.3 MNIST Autoencoder Baseline (Keras)

8.3.1 Dataset and Preprocessing

The MNIST dataset is used for unsupervised learning, consisting of 28 × 28 grayscale digit images. Images are:

- Flattened into 784-dimensional vectors
- Normalized to the range **[0, 1]**

This preprocessing matches the decoder's sigmoid output activation.

8.3.2 Autoencoder Architecture

The Keras autoencoder is intentionally kept simple to mirror the custom implementation:

- **Encoder:** Dense layer reducing 784 → 64 with ReLU activation
- **Decoder:** Dense layer expanding 64 → 784 with sigmoid activation

This single-hidden-layer design emphasizes representation learning rather than architectural complexity.

8.3.3 Training Configuration

- **Optimizer:** SGD
- **Learning rate:** 0.2
- **Loss function:** MSE
- **Batch size:** 128
- **Epochs:** 500

Validation is performed on the test set to monitor reconstruction performance

9. Comparison

name	lib		keras	
models	XOR	Autoencoder	XOR	Autoencoder
loss	0.00103	0.004957	0.000111	0.007425
Training time	10 seconds	3121.76 seconds	216.3284 seconds	822.90 seconds

Repo link :- https://github.com/aymanGit0/CI_Project/tree/main