Neural Network Library - Project Report

1. Library Design and Architecture

1. **Library Design and Architecture**

The neural network library was designed to be modular, readable, and easily extensible, similar to simplified versions of PyTorch or Keras. The core idea is to separate every function of a neural network into clearly defined components:

2. **Layer Abstraction (Layer Base Class)**

All layers in the library inherit from a simple Layer base class. This ensures every layer implements:

forward(x) — computes the output of the layer

backward(grad_output) — computes gradients and sends gradient backward

(optional) update(optimizer) — updates weights if the layer has parameters

This abstraction allows stacking any sequence of layers inside a Sequential model.

3. **Dense Layer (Dense)**

The Dense (fully-connected) layer is the main trainable component of the network.

Each Dense layer stores:

Weights W

Biases b

Gradients dW, db

Input cache (needed for backward pass)

Forward pass:

$y=xW+b$

Backward pass computes:

Gradient wrt weights

Gradient wrt biases

Gradient wrt inputs

This structure mirrors how professional deep learning frameworks implement linear layers.

4. **Activation Functions**

Activations are implemented as separate layers (e.g., Tanh, Sigmoid). They do not have weights, only formulas for:

forward activation

backward derivative

This modularity allows replacing activations without modifying the network architecture.

5. **Loss Functions (MSELoss)**

Loss functions are isolated from the model and only compute:

forward(y_true, y_pred)

backward() → gradient wrt predictions

This design keeps the model focused on learning rather than evaluating error.

6. **Optimizer (SGD)**

The SGD optimizer handles weight updates:

$W = W - \eta \cdot dW$

7. **Model Container (Sequential)**

The Sequential class stores a list of layers and manages:

Forward propagation through all layers

Backward propagation from last to first layer

Calling each layer's optimizer update

Running training loops with epochs

It behaves similarly to tf.keras.Sequential.

8. **Directory Structure**

The library is organized inside the AROGO/ folder:

AROGO/ │── layers.py │── activations.py │── losses.py │── optimizer.py └── network.py

This clean separation makes the codebase:

Easier to maintain

Easy to extend

Structured like real machine-learning libraries

Summary

The core principles of the library are:

Modularity (each component is independent)

Clarity (easy to read and understand)

Extensibility (easy to add new layers/activations)

Realistic design inspired by PyTorch/Keras

2. Gradient Checking

1. **Analytical Gradients:**
    - Computed using the `backward()` methods of our layers.
2. **Numerical Gradients:**
    - Approximated with the formula:
      [\frac{\partial L}{\partial W} \approx \frac{L(W + \epsilon) - L(W - \epsilon)}{2 \epsilon}]
    - Here, (\epsilon) is a very small number (default (1e-5)).
3. **Comparison:**
    - We calculate the maximum difference between analytical and numerical gradients.
    - If the difference is very small (e.g., less than (1e-6)), our backpropagation is likely correct.
4. **Results:**

Analytical: [[-0.00232373 -0.00274231 -0.00977339 0.01306674] [-0.00434791 0.01133584 0.002093 0.01870656]] Numerical: [[-0.00232373 -0.00274231 -0.00977339 0.01306674] [-0.00434791 0.01133584 0.002093 0.01870656]] Max Diff: 2.9750407866402373e-12

3. XOR Experiment

Epoch 0, Loss: 0.2539001108002241 Epoch 2000, Loss: 0.006986212600061806 Epoch 4000, Loss: 0.002426433168836391 Epoch 6000, Loss: 0.00142500829139407 Epoch 8000, Loss: 0.0009993669268076304 Predictions after training: [[0.01332864] [0.97127211] [0.97049072] [0.03451367]]

4. Analysis of the autoencoder's reconstruction quality

1-Loss Trend Over Epochs :

Both training loss and validation loss decrease smoothly from epoch 100 to epoch 1500:

Train loss: 0.068 → 0.025

Validation loss: 0.068 → 0.025

This shows that the autoencoder is continuously improving its reconstruction quality as training progresses.

2- Training vs Validation Loss Behavior :

A key observation is that:

Training loss ≈ Validation loss at every checkpoint

The difference between them is extremely small (≈ 0.00003)

What this means:

The model generalizes very well

There is no overfitting

The encoder is learning features that work equally well on unseen data

This is ideal behavior for an unsupervised model.

3- Stability of Learning :

The loss curve:

Is smooth

Has no sudden jumps

Shows steady convergence

This indicates:

The learning rate is well chosen

Gradient updates are stable

The training process is numerically sound

4- Final Reconstruction Quality :

At epoch 1500:

Train Loss ≈ 0.025

Validation Loss ≈ 0.025

For MNIST images normalized to [0, 1], this is considered a low reconstruction error, meaning:

Digit structure is preserved

Major strokes are reconstructed correctly

Minor blurring is expected and normal

5- Interpretation of the Latent Space :

Since reconstruction quality keeps improving:

The 64-dimensional latent space is sufficient

Important digit features (shape, strokes, orientation) are captured

Noise and irrelevant pixel variations are discarded

This explains why the latent features later achieve high SVM classification accuracy.

6- Conclusion :

The autoencoder demonstrates stable and effective learning, with steadily decreasing and closely aligned training and validation losses across 1500 epochs. The absence of divergence between the two curves confirms strong generalization and no overfitting. The final low reconstruction error indicates that the encoder successfully learns compact and informative latent representations, preserving essential digit structure while achieving efficient compression.

5. Analysis of SVM Classification Results and Latent Space Quality :

The Support Vector Machine (SVM) classifier was trained on the latent representations produced by the encoder of the autoencoder. The encoder maps each 784-dimensional MNIST image into a compact 64-dimensional latent space, which is expected to capture the most meaningful and discriminative features of the input data.

1- Overall Classification Performance :

The SVM achieved the following accuracies:

Training Accuracy: 96.78%

Validation Accuracy: 95.52%

Test Accuracy: 95.37%

The close alignment between training, validation, and test accuracies indicates that the classifier generalizes well and is not overfitting. This consistency strongly suggests that the latent features learned by the encoder are stable and meaningful across unseen data.

2- Quality of the Learned Latent Space :

The high classification accuracy demonstrates that the encoder learned a well-structured latent space with the following properties:

Class separability: Digits belonging to the same class are clustered closely in latent space, while different classes are well separated.

Low-dimensional efficiency: Despite reducing the input from 784 dimensions to only 64, the latent space preserves enough information to distinguish between digits accurately.

Noise reduction: The autoencoder removes irrelevant pixel-level noise and retains only the most important patterns (strokes, shapes, digit structure), making classification easier for the SVM.

3- Confusion Matrix Interpretation :

The confusion matrix shows that most predictions lie on the diagonal, indicating correct classification. Misclassifications mainly occur between visually similar digits such as:

4 and 9

5 and 3

8 and 9

These errors are expected due to similarities in handwritten digit shapes. Importantly, the number of such errors is small, confirming that the latent space captures discriminative visual features effectively.

4- Classification Report Insights :

Precision and Recall values are high (≈ 0.95) across most classes.

Balanced performance across digits indicates that no class dominates the latent space representation.

The strong F1-scores reflect both accurate and consistent predictions.

5- Conclusion :

The SVM results confirm that the encoder learned a high-quality latent representation that is:

Compact

Discriminative

Robust

Suitable for downstream supervised tasks

This demonstrates the effectiveness of the autoencoder as an unsupervised feature extractor, where meaningful features are learned without using labels, yet perform exceptionally well when combined with a supervised classifier.

6.Challenges Faced and Lessons Learned :

Challenges Faced :

1- Training Stability in the Custom Library: Implementing the autoencoder using a custom NumPy-based neural network library required careful tuning of learning rates and weight initialization. High learning rates caused unstable training, while low values led to very slow convergence.

2- Validation Handling : Unlike TensorFlow, validation had to be implemented manually in Part 2. Ensuring that validation data was not used for weight updates required strict separation between training and validation forward passes.

3- Long Training Time : Training the autoencoder for a large number of epochs (up to 1000–1500) on MNIST using NumPy was computationally expensive and significantly slower than TensorFlow due to the lack of GPU acceleration and optimized backend operations.

4- Latent Space Interpretation : Since the autoencoder is unsupervised, evaluating whether the latent space was “good” was not straightforward. This required an additional supervised task (SVM classification) to objectively assess feature quality.

5- Digit Confusion (e.g., 4 vs 9) : Some digits with similar handwritten shapes were frequently confused, highlighting the inherent ambiguity in the dataset rather than an error in implementation.

Lessons Learned

1- Importance of Validation : Validation is essential even in unsupervised learning. Monitoring validation loss helped detect overfitting and ensured that the autoencoder generalized beyond the training data.

2- Power of Unsupervised Feature Learning : Although labels were not used during autoencoder training, the learned latent features proved highly effective for classification, achieving over 95% test accuracy with an SVM.

3- Latent Space Quality Matters More Than Model Complexity : Reducing input dimensionality from 784 to 64 did not significantly degrade performance. This demonstrated that well-learned representations are more important than raw input dimensionality.

4- Scaling Is Fundamental : Feature scaling was critical for SVM performance. It improved numerical stability and convergence without adding information or leaking labels.

5- Visualization Is a Powerful Diagnostic Tool : Visualizing reconstruction results helped confirm that the autoencoder learned meaningful structures rather than memorizing noise.

Final Insight :

This project demonstrated that building neural networks from scratch enhances conceptual understanding, while modern frameworks like TensorFlow excel in efficiency and scalability. Combining both approaches provided both theoretical depth and practical performance.