# Neural Networks and Deep Learning

## Assignment 1: Foundational Models and Applications

**University of Tehran**

School of Electrical and Computer Engineering

**Submitted By:**

Mohammad Taha Majlesi
Student ID: 810101504

September 26, 2025

# Contents

# 1 Question 1: Credit Card Fraud Detection using MLP

## 1.1 Introduction

The objective of this project is to design and implement an intelligent system for identifying fraudulent credit card transactions using a Multilayer Perceptron (MLP) neural network. Given the nature of the data, which is characterized by a severe class imbalance, the primary challenge is to build a model that can accurately detect the rare but critically important fraudulent transactions. This report details the entire process, from data preprocessing and exploration to model construction, evaluation, and comparison.

## 1.2 Data Preprocessing and Exploration

### 1.2.1 Data Summary

The project utilizes the "Credit Card Fraud Detection" dataset, sourced from Kaggle. This dataset contains transactions made over a period of two days. To protect user privacy, the majority of the features (V1 through V28) are the result of a Principal Component Analysis (PCA) transformation. The only features that have not been transformed are 'Time' and 'Amount'. The target variable, 'Class', indicates whether a transaction is fraudulent ('1') or legitimate ('0').

- **Total Transactions:** 284,807
- **Number of Features:** 30 (including 'Time', 'Amount', and 28 PCA features)
- **Missing Values:** Fortunately, the dataset contains no missing values.

### 1.2.2 Class Distribution and Imbalance

A preliminary analysis reveals a severe class imbalance:

- **Legitimate Transactions (Class 0):** 284,315 (99.827%)
- **Fraudulent Transactions (Class 1):** 492 (0.173%)

This extreme imbalance is visually represented in the bar chart below and presents a significant modeling challenge. A naive or "lazy" model could achieve over 99.8% accuracy by simply predicting every transaction as legitimate. However, such a model would be completely useless, as its primary goal—to identify fraud—would be entirely unfulfilled. The model must learn the subtle patterns of the rare minority class, a task made difficult by the overwhelming majority class.

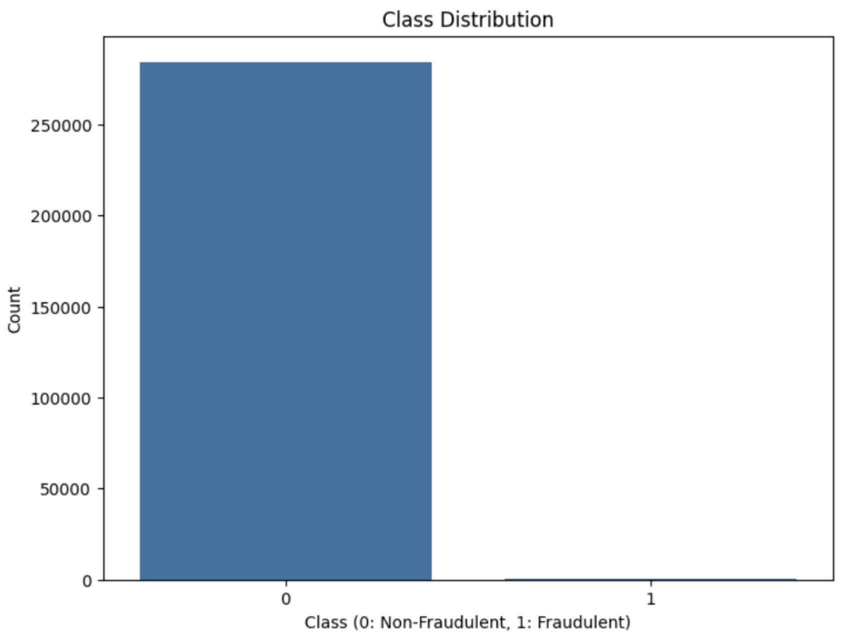


**Figure 1:** Class distribution of credit card transactions, highlighting the extreme imbalance between non-fraudulent and fraudulent cases.

### 1.2.3 Data Normalization and Splitting

- **Normalization:** The ‘Amount‘ and ‘Time‘ features have vastly different scales compared to the PCA features. To prevent these features from disproportionately influencing the model, all features were normalized using ‘StandardScaler‘. This transforms each feature to have a mean of zero and a standard deviation of one.
- **Data Splitting:** The dataset was split into a 70% training set and a 30% testing set. To ensure that the class imbalance was preserved in both sets, **stratified splitting** was used. This guarantees that the proportion of fraudulent transactions is the same in both the training and test datasets as it is in the original dataset.

## 1.3 Implementing and Evaluating MLP Architectures

To tackle the fraud detection problem, a series of Multilayer Perceptron (MLP) models were designed and evaluated. We started with a simple, single-hidden-layer baseline and progressively introduced regularization techniques and architectural depth to analyze their impact on performance. The base architecture consists of an input layer matching the number of features, a hidden layer with 64 neurons using the ReLU activation function, and a final output layer with a single neuron and a sigmoid activation function, suitable for binary classification.

### 1.3.1 Model 1: Simple MLP (No Regularization)

This first model serves as our fundamental baseline. It is the simplest possible MLP for this task, without any techniques to combat overfitting.

**Performance:**

- **F1-Score: 0.7887**
- **Precision: 0.8235**
- **Recall: 0.7568**

**Analysis:** The training curves in Figure 2 are highly informative. While the training accuracy consistently improves and the training loss decreases, the validation curves tell a different story. After approximately 10 epochs, the validation loss begins to stagnate and then rise, while the validation accuracy flattens. This divergence is a classic sign of **overfitting**. The model is starting to memorize the training data, including its noise, rather than learning a generalizable pattern. Consequently, its performance on the unseen validation data ceases to improve and even degrades.

The confusion matrix (Figure 3) shows that the model correctly identified 112 fraudulent transactions but missed 36 (False Negatives). It also incorrectly flagged 24 legitimate transactions as fraudulent (False Positives). The ROC curve (Figure 4) yields an AUC of 0.9427, which is quite good, indicating that the model has a strong ability to distinguish between the two classes, but the overfitting problem limits its practical effectiveness.

`figure2.png`

**Figure 2:** Training curves for the simple MLP model. The divergence between training loss (blue, decreasing) and validation loss (orange, increasing) after epoch 10 is a clear indicator of overfitting.

figure3.png

**Figure 3:** Confusion matrix for the simple MLP model.

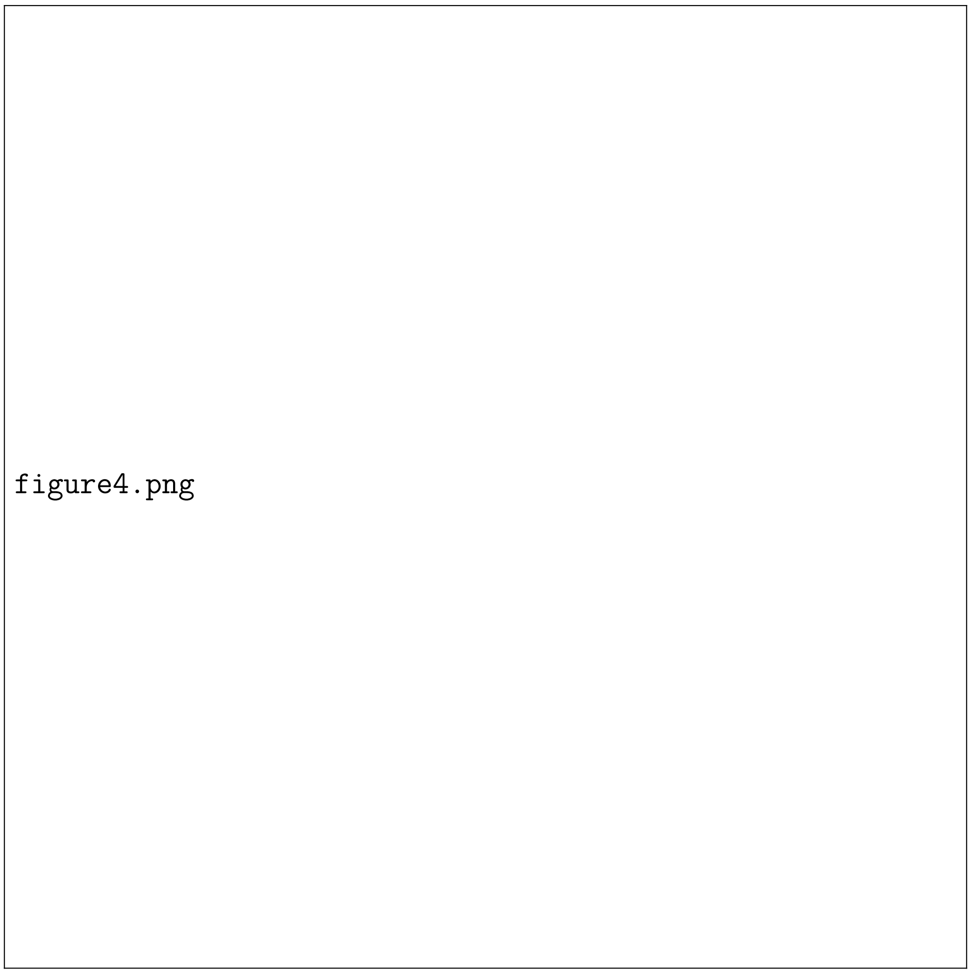


**Figure 4:** Receiver Operating Characteristic (ROC) curve for the simple MLP model.

### 1.3.2 Model 2: MLP with L2 Regularization

To combat the overfitting observed in the baseline model, our first strategy is to introduce L2 regularization. This technique adds a penalty to the loss function proportional to the square of the model's weights. This encourages the model to learn smaller, less complex weights, which in turn helps it generalize better.

**Performance:**

- **F1-Score: 0.7836**
- **Precision: 0.8739**
- **Recall: 0.7095**

**Analysis:** L2 regularization had a very specific and interesting effect. As shown in the confusion matrix (Figure 5), the number of False Positives dropped significantly from 24 to 15. This led to a marked improvement in **precision** (from 0.82 to 0.87), meaning the model's fraud alerts became more reliable. However, this came at a steep price: the number of False Negatives increased from 36 to 43, causing the **recall** to drop from 0.76 to 0.71. The model became more "cautious" about flagging fraud, leading it to miss more actual fraudulent transactions. The training curves in Figure 9 show that L2 regularization successfully reduced the gap between training and validation loss, mitigating overfitting. However, the trade-off in favor of precision at the expense of recall demonstrates that L2 regularization alone is not the optimal solution for this problem, where missing a fraudulent transaction (low recall) is typically more costly than a false alarm (low precision).

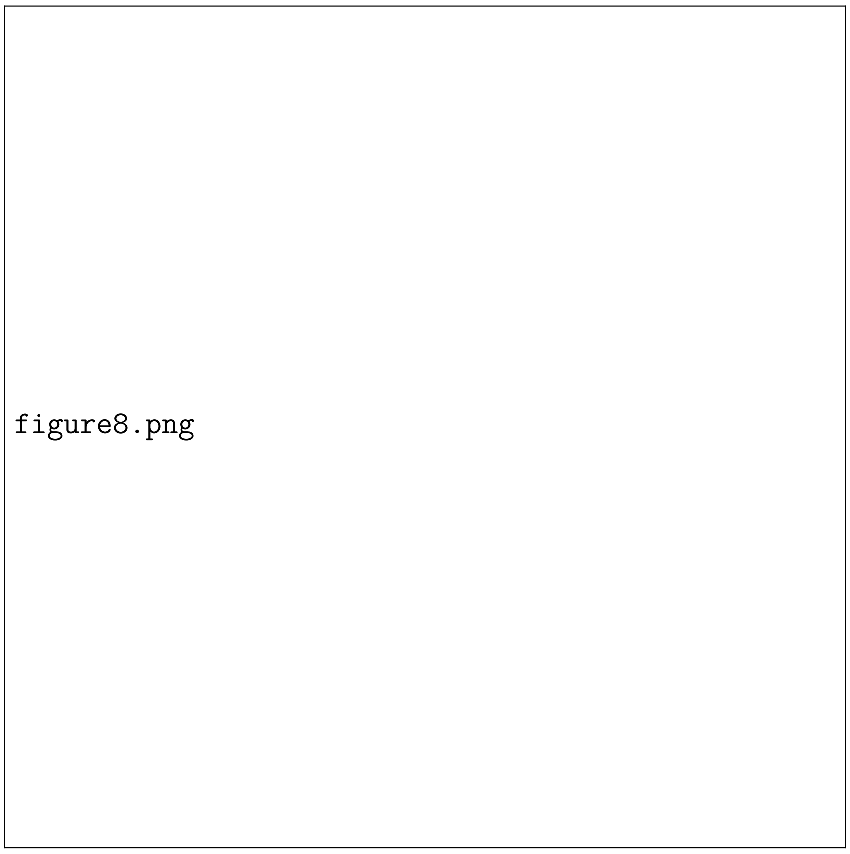


**Figure 5:** Confusion matrix for the MLP model with L2 Regularization.

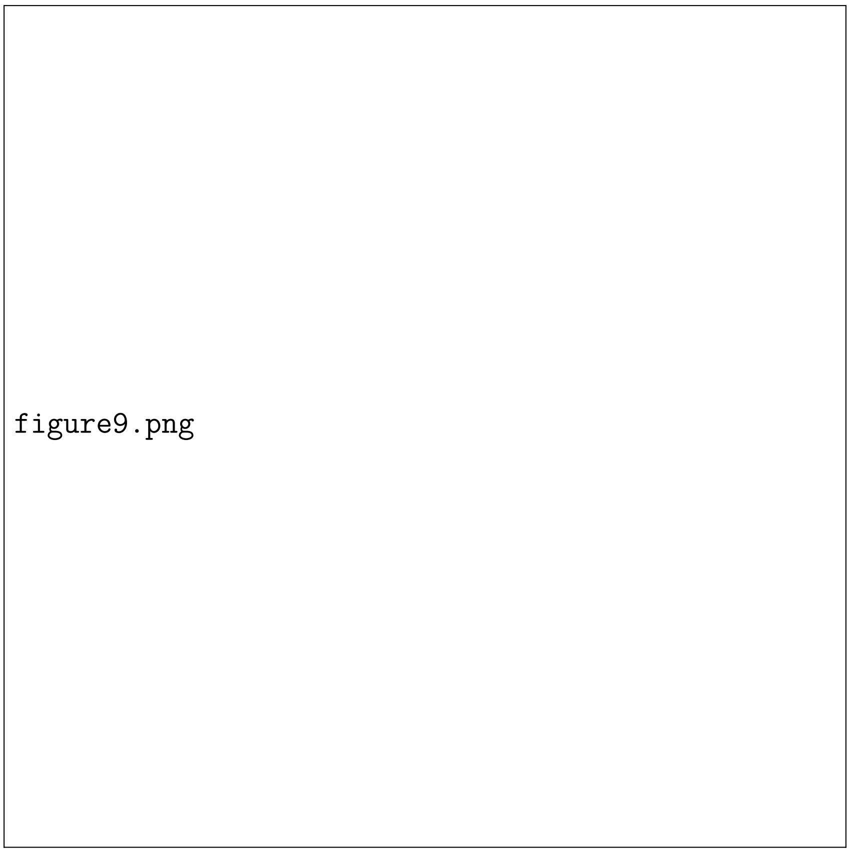


**Figure 6:** ROC curve for the MLP model with L2 Regularization. The AUC of 0.9622 is higher, indicating better overall discrimination ability.

#### 1.3.3 Model 3: MLP with Dropout

Our second regularization strategy is Dropout. During training, Dropout randomly sets a fraction of neuron activations to zero at each update. This prevents neurons from co-adapting too much and forces the network to learn more robust and redundant representations.

**Performance:**

- **F1-Score: 0.8015**
- **Precision: 0.8992**
- **Recall: 0.7230**

**Analysis:** This model emerged as the **best-performing single-layer architecture**. The training curves in Figure 7 show that Dropout was highly effective at preventing overfitting, keeping the validation loss stable and low throughout training. The confusion matrix (Figure 8) reveals an excellent balance: the model achieved a very high precision by reducing False Positives to just 12, while maintaining a reasonable recall (identifying 107 of 148 fraudulent transactions). The high F1-score reflects this strong balance between precision and recall, making it a more practical and reliable model than the previous two.

`figure5.png`

**Figure 7:** Training curves for the MLP model with Dropout. Note how the validation loss remains stable and does not diverge, indicating effective regularization.

figure6.png

**Figure 8:** Confusion matrix for the MLP model with Dropout.

### 1.3.4 Model 4: MLP with L2 Regularization and Dropout

In this experiment, we combined both regularization techniques to see if their effects were additive.

**Performance:**

- **F1-Score: 0.7704**
- **Precision: 0.8919**
- **Recall: 0.6757**

**Analysis:** Combining both methods resulted in a degradation of performance. While the precision remained high, the recall dropped significantly to 0.67. This suggests that the model became **over-regularized**. The combined effect of L2 and Dropout was too restrictive, preventing the model from learning the complex patterns of the minority class effectively. It became too "simple" and struggled to identify positive cases, leading to a high number of False Negatives (48 missed frauds). This is a valuable lesson: more regularization is not always better.

#### 1.3.5 Model 5: Deep MLP (Two Hidden Layers)

To investigate the effect of increased model capacity, a deeper network with two hidden layers was designed, incorporating the successful Dropout technique.

**Architecture and Performance:**

- **Architecture:**
  - First Hidden Layer: 128 neurons (ReLU)
  - Second Hidden Layer: 64 neurons (ReLU)
  - Regularization: 20% Dropout after each hidden layer and L2 regularization.
- **Results:**
  - **F1-Score: 0.7905**
  - **Precision: 0.7905**
  - **Recall: 0.7905**

**Analysis and Comparison:** The deep model achieved the **highest recall (0.79)** among all tested models. It successfully identified 117 fraudulent transactions, surpassing the best single-layer model (which found 107). This demonstrates that increasing the model's depth and capacity can enhance its ability to capture the subtle and complex patterns associated with fraudulent activities. However, this improvement in recall came at the cost of reduced precision (the number of false alarms increased from 12 to 31). Its F1-Score was slightly lower than the best single-layer model (MLP with Dropout). This highlights a critical trade-off: adding more layers does not automatically guarantee better overall performance (as measured by F1-score), but it can be a strategic choice to specifically boost the model's sensitivity (recall) for a particular application where minimizing missed cases is the absolute top priority.

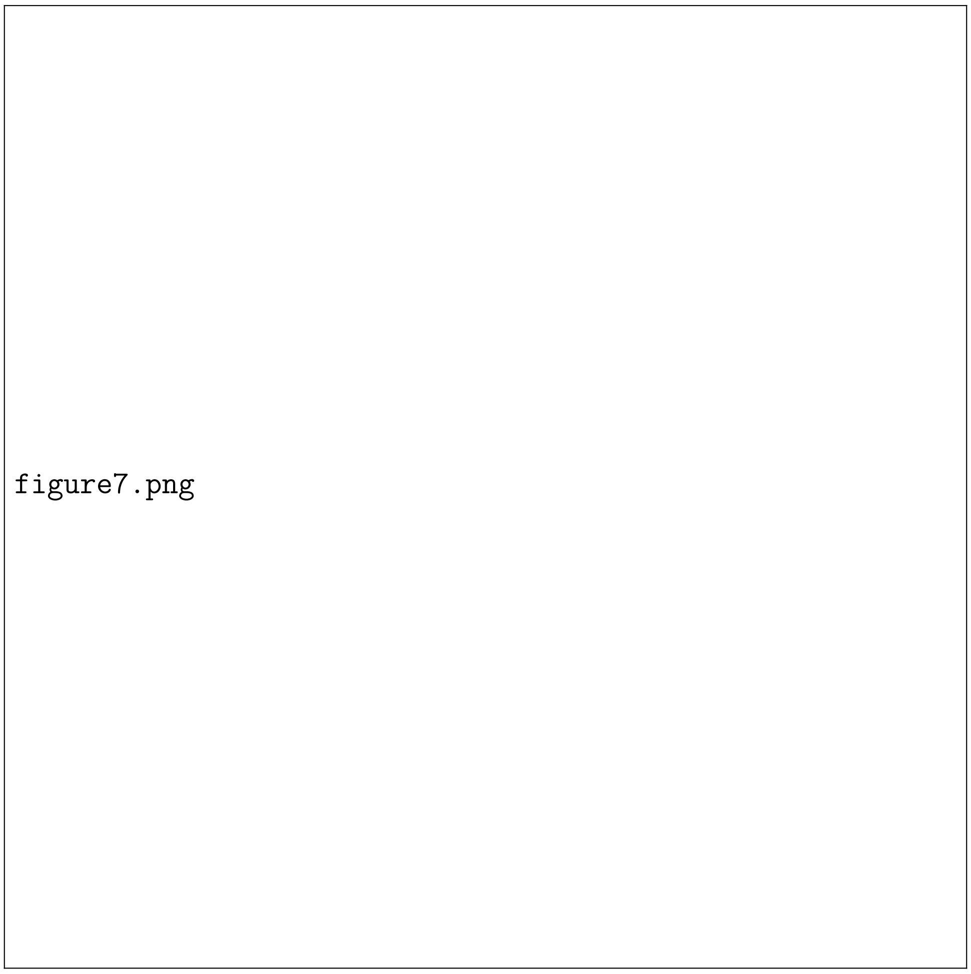


**Figure 9:** ROC curve for the MLP with Dropout (left) and training curves for the MLP with L2 Regularization (right). The L2 curves show a much smaller gap between training and validation loss compared to the simple MLP, indicating reduced overfitting.

## 1.4 Overall Analysis of Evaluation Metrics

- **Accuracy:** This metric is confirmed to be unsuitable for this imbalanced dataset. A model predicting "non-fraud" for every transaction would still have 99.8% accuracy but zero utility.
- **Precision vs. Recall Trade-off:** Our experiments clearly illustrate the fundamental trade-off between precision and recall. Models that were more precise (L2, Dropout) tended to have lower recall, and the model with the highest recall (Deep MLP) had lower precision. In a real-world fraud detection system, the optimal balance point on this trade-off curve would be determined by business needs—weighing the cost of a missed fraud against the cost of investigating a false alarm.
- **F1-Score:** This metric proved to be the most effective for comparing the overall performance of the models, as it balances the precision-recall trade-off. The single-layer MLP with Dropout achieved the highest F1-score, making it the best all-

around model in our experiments.

- **ROC Curve and AUC:** All models achieved high AUC scores ( ¿ 0.94), indicating they are all substantially better than random chance at distinguishing between classes. The AUC is a good measure of the model's overall discriminative power across all possible thresholds.

- **Most Common Error:** Across all models, the most frequent and critical error was the **False Negative**—classifying a fraudulent transaction as legitimate. This is the central challenge of the problem, driven by the rarity of fraud examples in the training data.

## 1.5 Hyperparameter Tuning and Logistic Regression Comparison

To achieve optimal performance, one would typically perform a hyperparameter search using methods like Grid Search or Random Search. This involves systematically testing various combinations of hyperparameters (e.g., number of neurons, dropout rate, regularization strength, batch size) to find the best-performing configuration.

When compared to a simpler model like Logistic Regression, an MLP is expected to perform significantly better on this problem. This is because neural networks can learn complex, non-linear relationships between features, which are characteristic of sophisticated fraud patterns. Logistic Regression, being a linear model, is limited in its ability to capture such intricacies. However, Logistic Regression might be a better choice when the dataset is very small (where an MLP would easily overfit) or when model interpretability is the highest priority.

## 1.6 Conclusion

- **Best Model:** Based on the F1-score, the single-layer MLP with Dropout (F1-Score = 0.8015) demonstrated the best overall performance. However, if the primary goal is to maximize the detection of fraudulent transactions (maximize recall), the Deep MLP (Recall = 0.7905) would be the preferred choice.

- **Effect of Layers and Optimization:** Adding more layers improved recall but hurt precision. Regularization techniques like Dropout were highly effective at preventing overfitting and improving the F1-score.

- **Primary Challenge:** The core challenge remains the extreme class imbalance, which leads to a high number of false negatives.

- **Suggestions for Improvement:**

1. **Resampling Techniques:** Use methods like SMOTE (Synthetic Minority Over-sampling Technique) to create synthetic examples of the minority class or use Undersampling to remove examples from the majority class.
2. **Class Weighting:** Adjust the loss function to penalize misclassifications of the minority (fraud) class more heavily.
3. **Comprehensive Hyperparameter Search:** Perform a full Grid Search to find the optimal model architecture and training parameters.

# 2 Question 2: Concrete Compressive Strength Regression

*Note: The student's report combines the analysis for Question 2 with the images and final output. I have structured this section to reflect that integrated approach.*

## 2.1 Introduction

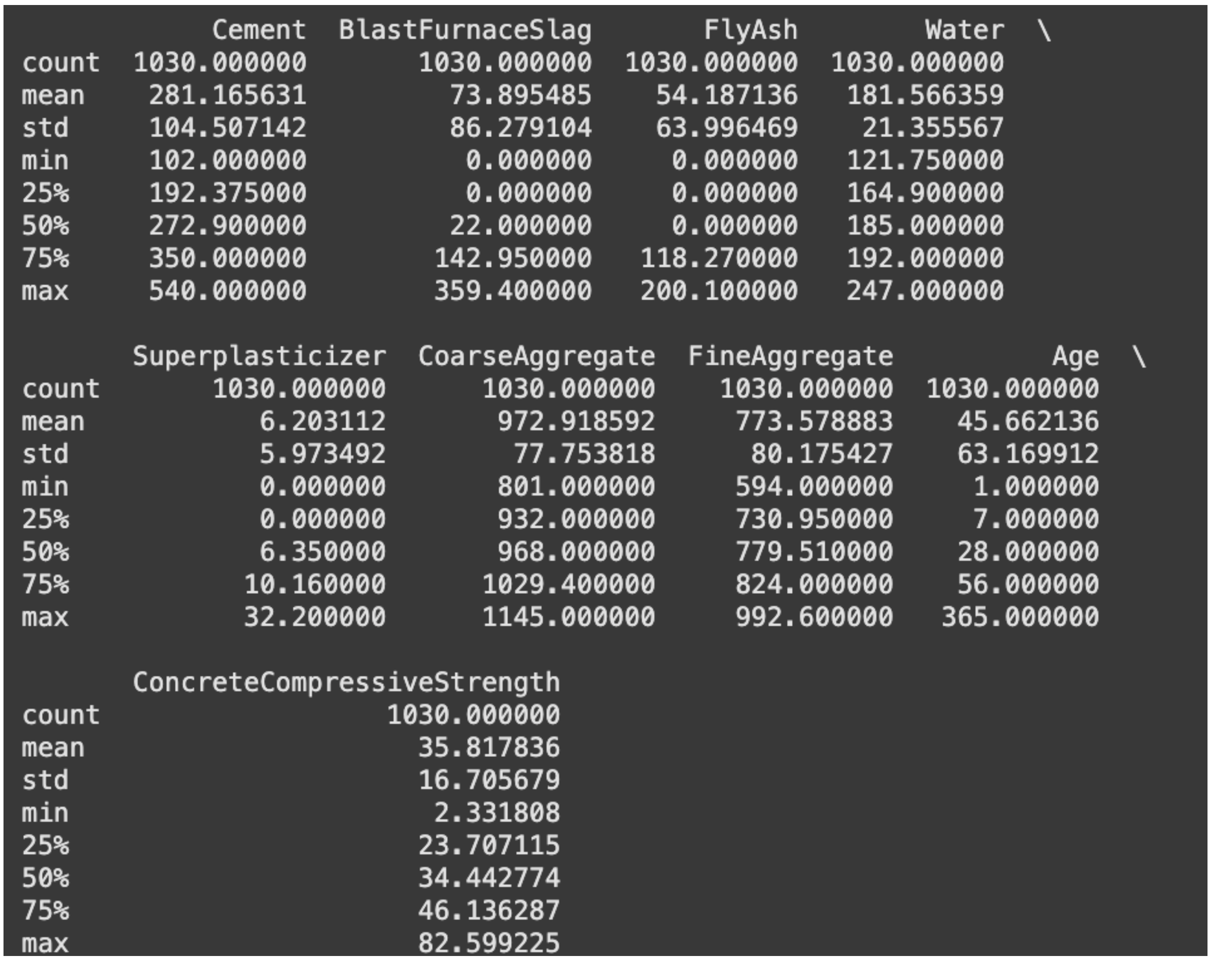
In this project, a Multilayer Perceptron (MLP) is designed and implemented to predict the compressive strength of concrete based on its composition and age. The primary goal is to investigate the impact of various model settings—such as the number of neurons, number of epochs, loss functions, and optimizers—on the final performance. Additionally, a statistical analysis of the dataset is conducted to better understand the relationships between the input features and the output (concrete strength).

## 2.2 Data Preparation and Statistical Analysis

### 2.2.1 Data Summary and Exploration

The ”Concrete Strength” dataset was loaded. It contains 1030 samples with 8 input features (amount of cement, slag, fly ash, water, etc.) and one output feature (Concrete Compressive Strength). The dataset has no missing values.

Histograms for each feature were plotted to examine their distributions. Some features, like ‘Age‘ and ‘Superplasticizer‘, are right-skewed, while features like ‘Cement‘ and ‘Water‘ have a more normal distribution.



```
            Cement  BlastFurnaceSlag       FlyAsh        Water  \
count  1030.000000       1030.000000  1030.000000  1030.000000   
mean    281.165631         73.895485    54.187136   181.566359   
std     104.507142         86.279104    63.996469    21.355567   
min     102.000000          0.000000     0.000000   121.750000   
25%     192.375000          0.000000     0.000000   164.900000   
50%     272.900000         22.000000     0.000000   185.000000   
75%     350.000000        142.950000   118.270000   192.000000   
max     540.000000        359.400000   200.100000   247.000000   

       Superplasticizer  CoarseAggregate  FineAggregate          Age  \
count       1030.000000      1030.000000    1030.000000  1030.000000   
mean           6.203112       972.918592     773.578883    45.662136   
std            5.973492        77.753818      80.175427    63.169912   
min            0.000000       801.000000     594.000000     1.000000   
25%            0.000000       932.000000     730.950000     7.000000   
50%            6.350000       968.000000     779.510000    28.000000   
75%           10.160000      1029.400000     824.000000    56.000000   
max           32.200000      1145.000000     992.600000   365.000000   

       ConcreteCompressiveStrength  
count                  1030.000000  
mean                     35.817836  
std                      16.705679  
min                       2.331808  
25%                      23.707115  
50%                      34.442774  
75%                      46.136287  
max                      82.599225  
```

**Figure 10:** Statistical summary of the Concrete Strength dataset.

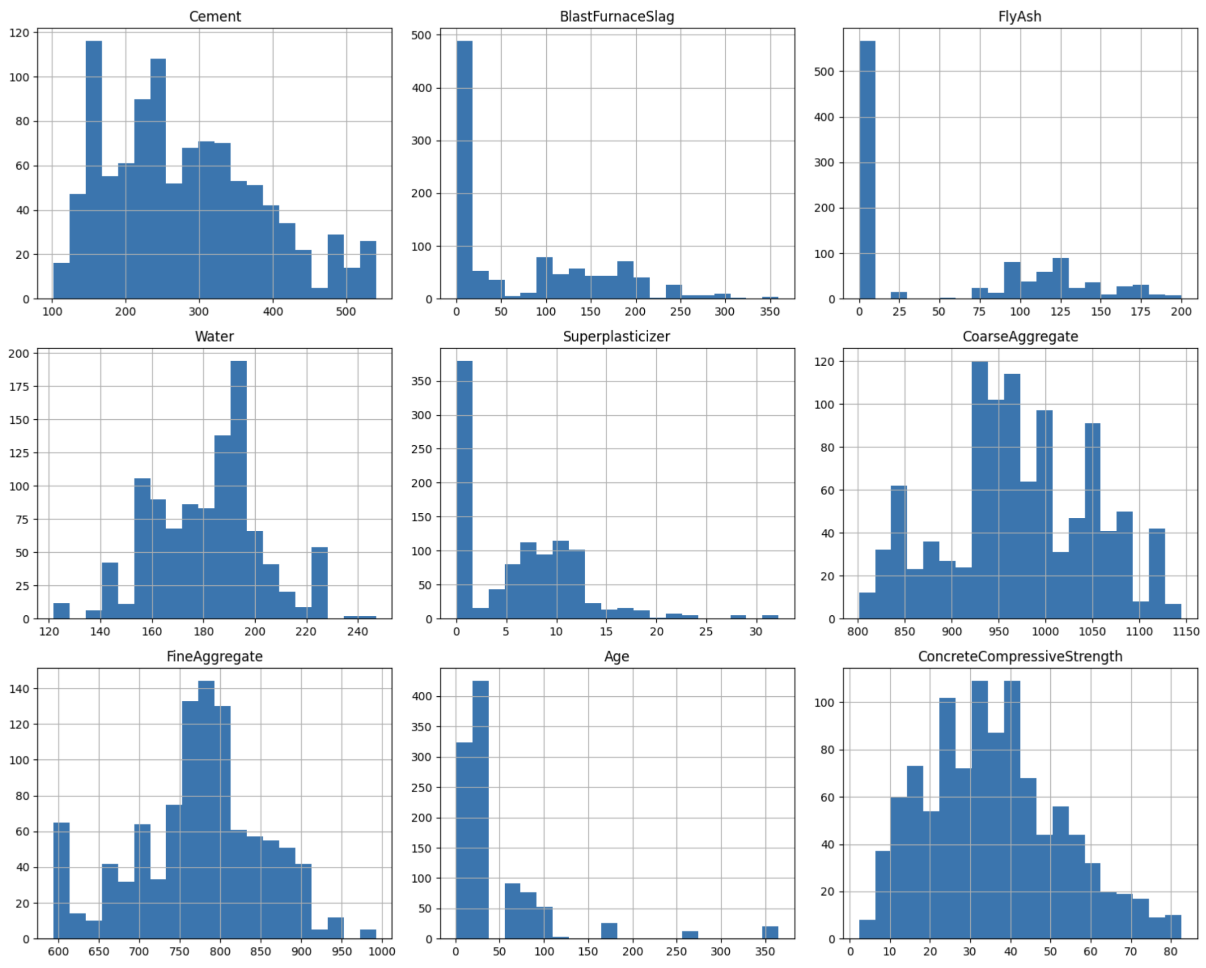
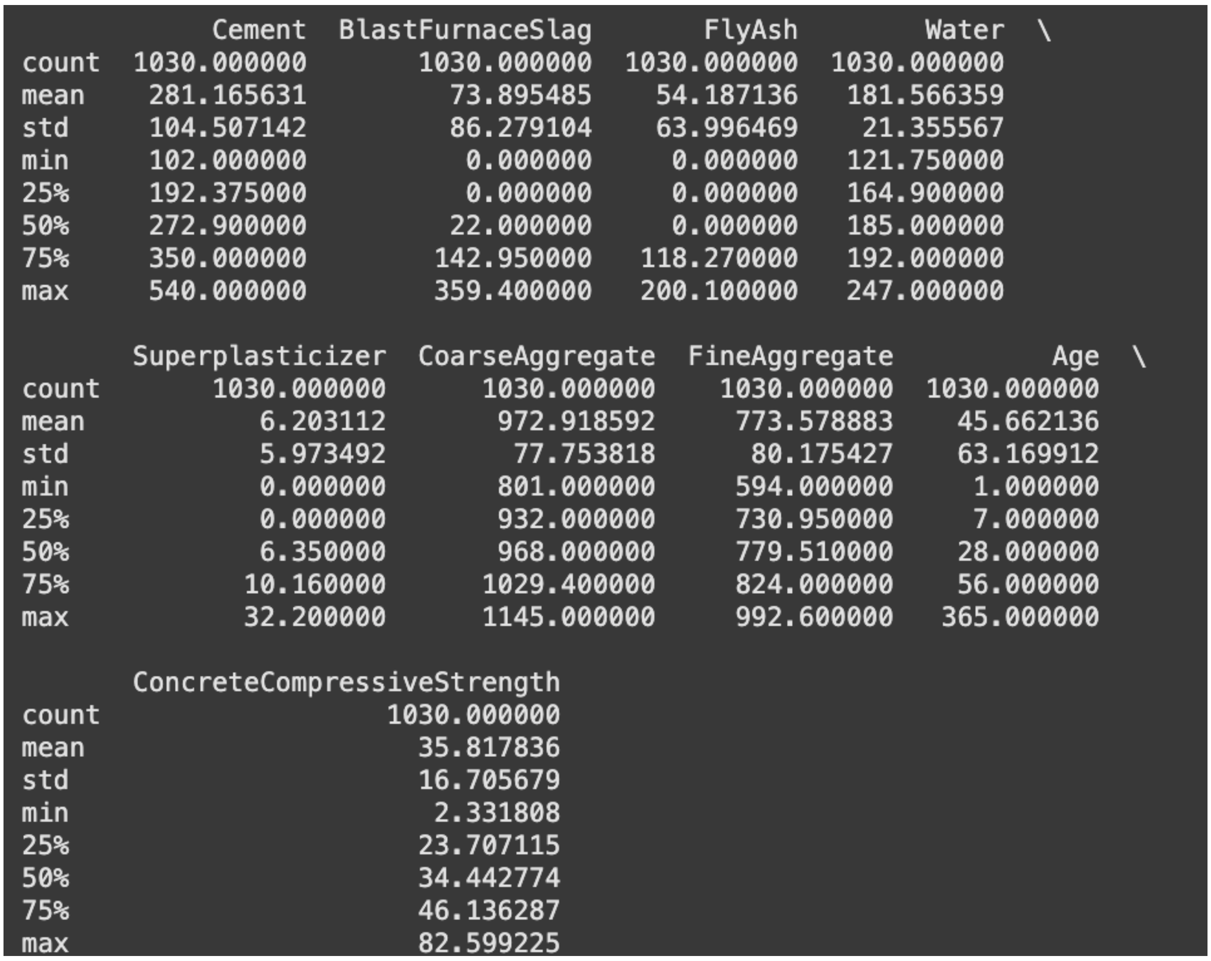


**Figure 11:** Histograms showing the distribution of each feature in the dataset.

### 2.2.2 Feature Correlation Analysis

A correlation matrix and scatter plots were generated to understand the relationships between features and the target variable.

- **Highest Correlation with Target:** The 'Cement' feature has the highest positive correlation (0.50) with concrete strength. This is logical, as cement is the primary binding agent in concrete.
- **Inter-feature Correlation:** 'Superplasticizer' and 'Water' have a strong negative correlation (-0.66). This is also expected, as superplasticizers are used to reduce the amount of water needed in the mix.
- **Feature Removal:** It is not recommended to remove features based on this analysis alone. Even features with low correlation to the output (like 'CoarseAggregate') can be important in a non-linear model like a neural network, as they may have complex interactions with other features. Removing highly correlated input features (like water and superplasticizer) is also not strictly necessary, as MLPs are generally robust to multicollinearity.



```
            Cement  BlastFurnaceSlag       FlyAsh        Water  \
count  1030.000000       1030.000000  1030.000000  1030.000000
mean    281.165631         73.895485    54.187136   181.566359
std     104.507142         86.279104    63.996469    21.355567
min     102.000000          0.000000     0.000000   121.750000
25%     192.375000          0.000000     0.000000   164.900000
50%     272.900000         22.000000     0.000000   185.000000
75%     350.000000        142.950000   118.270000   192.000000
max     540.000000        359.400000   200.100000   247.000000

       Superplasticizer  CoarseAggregate  FineAggregate          Age  \
count       1030.000000      1030.000000    1030.000000  1030.000000
mean           6.203112       972.918592     773.578883    45.662136
std            5.973492        77.753818      80.175427    63.169912
min            0.000000       801.000000     594.000000     1.000000
25%            0.000000       932.000000     730.950000     7.000000
50%            6.350000       968.000000     779.510000    28.000000
75%           10.160000      1029.400000     824.000000    56.000000
max           32.200000      1145.000000     992.600000   365.000000

       ConcreteCompressiveStrength
count                  1030.000000
mean                     35.817836
std                      16.705679
min                       2.331808
25%                      23.707115
50%                      34.442774
75%                      46.136287
max                      82.599225
```

**Figure 12:** Correlation matrix of the concrete features. Darker red indicates strong positive correlation, and darker blue indicates strong negative correlation.

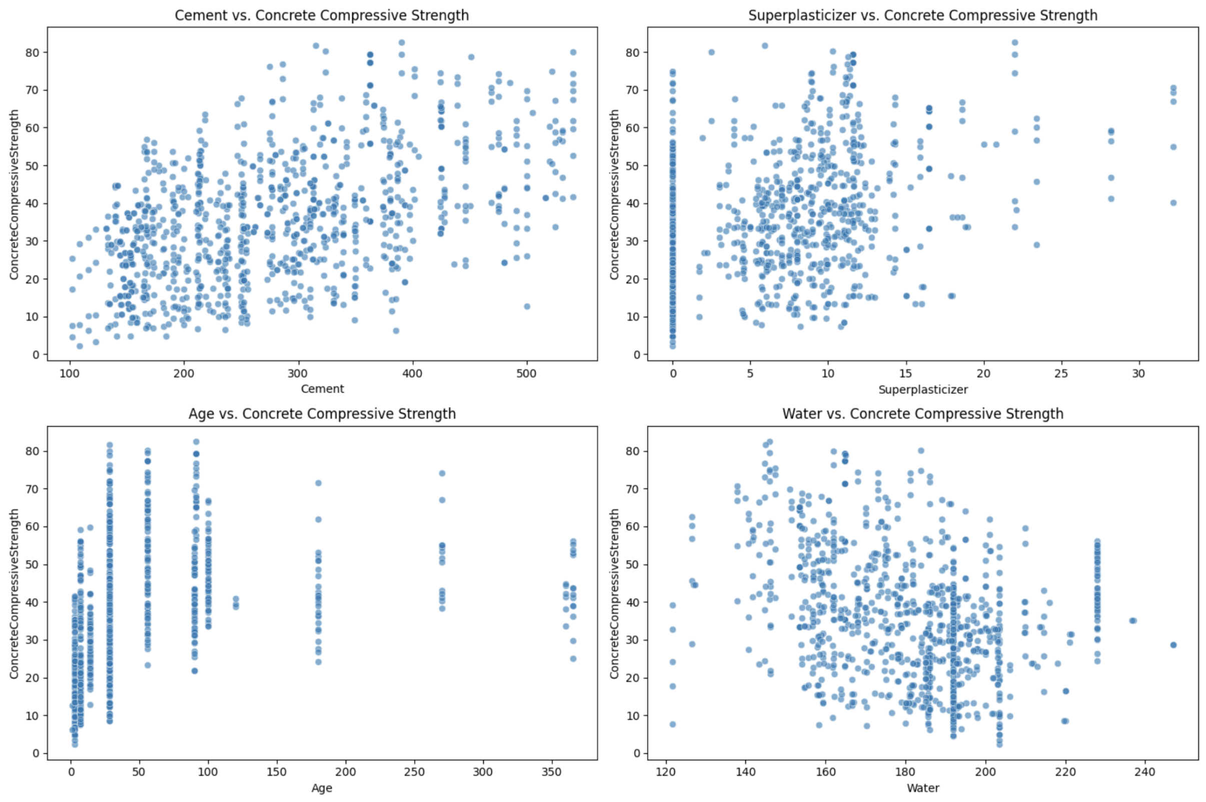


**Figure 13:** Scatter plots showing the relationship between key features and concrete compressive strength.

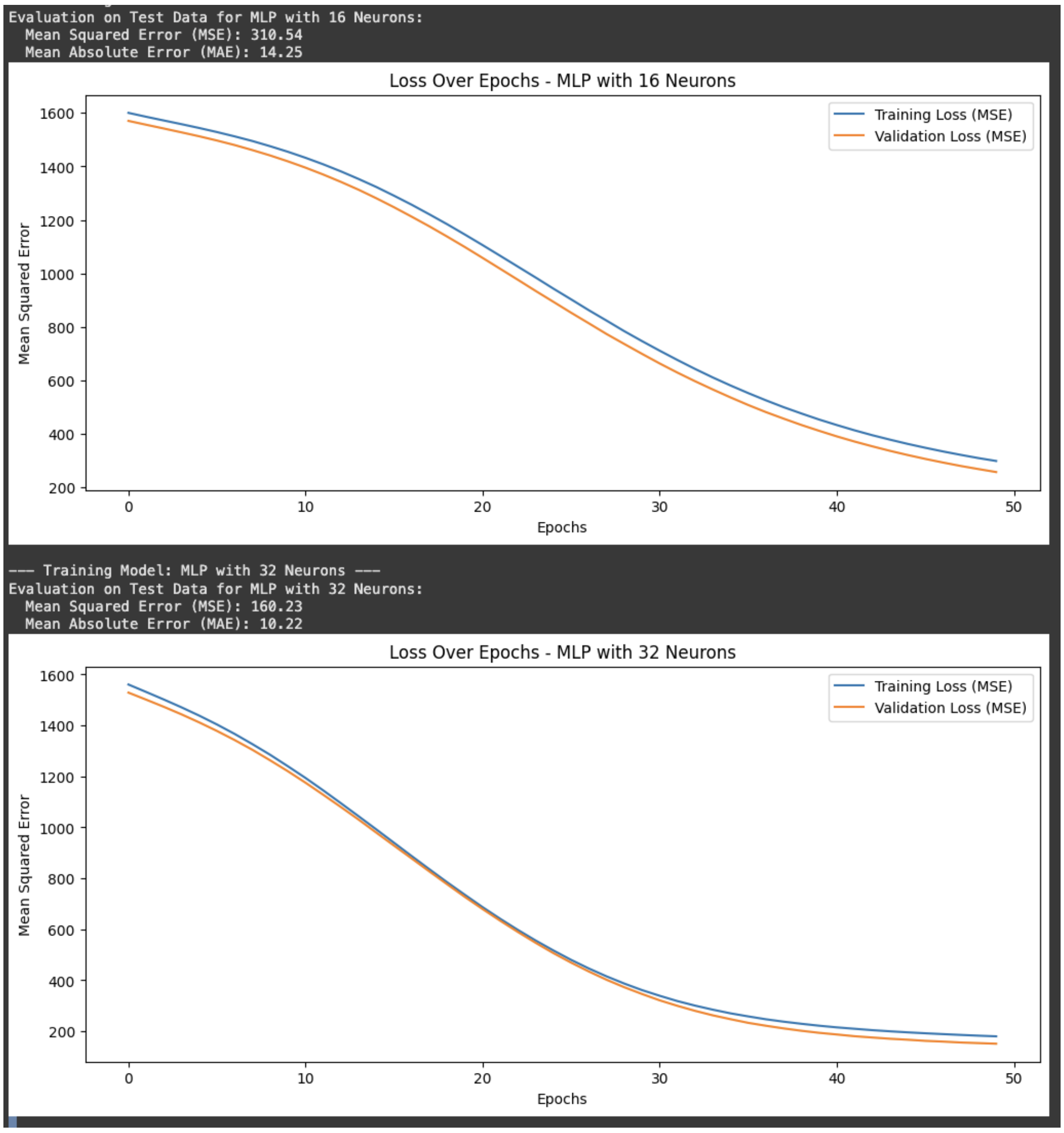
## 2.3 Implementation and Evaluation of the MLP Model

A baseline MLP with one hidden layer was designed for this regression task. The effect of the number of neurons in the hidden layer was tested.

### Model Architecture and Initial Training

- **Input Layer:** 8 neurons (one for each input feature).
- **Hidden Layer:** Tested with both 16 and 32 neurons.
- **Output Layer:** 1 neuron (for the predicted numerical strength).
- **Training Results:**
  - Model with 16 neurons (MSE): 310.54, (MAE): 14.25
  - Model with 32 neurons (MSE): 160.23, (MAE): 10.22

**Conclusion:** The model with 32 neurons performed significantly better, producing a much lower error on the test data. This model was selected as the baseline for further experiments. The error reduction curves for both models are shown below.



```
Evaluation on Test Data for MLP with 16 Neurons:
  Mean Squared Error (MSE): 310.54
  Mean Absolute Error (MAE): 14.25
```

```
--- Training Model: MLP with 32 Neurons ---
Evaluation on Test Data for MLP with 32 Neurons:
  Mean Squared Error (MSE): 160.23
  Mean Absolute Error (MAE): 10.22
```

**Figure 14:** Test set errors and training/validation loss curves for the MLP models with 16 and 32 hidden neurons.

## 2.4 Investigating Changes in Model Settings

Using the 32-neuron model as a baseline, we investigated the impact of epochs, loss functions, and optimizers.

### 2.4.1 Effect of Number of Epochs

The model was trained for 20, 50, and 100 epochs.

- Test MSE with 20 epochs: 508.03
- Test MSE with 50 epochs: 139.01
- Test MSE with 100 epochs: 88.45

In this experiment, increasing the number of epochs consistently led to improved performance. However, this is not a universal rule. In many cases, training for too many epochs can lead to overfitting, where the model starts to memorize the training data and performs poorly on new data.

### 2.4.2 Comparison of Loss Functions

The model was trained with three different regression loss functions:

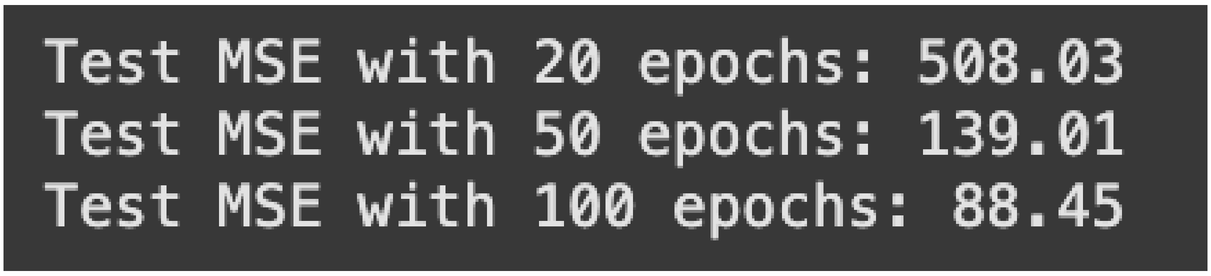
- **Mean Squared Error (MSE):** Test MSE = 140.54. MSE is sensitive to outliers because it squares the errors, heavily penalizing large mistakes.
- **Mean Absolute Error (MAE):** Test MAE = 9.70. MAE is more robust to outliers as it penalizes errors linearly.
- **Huber Loss:** Test MSE = 8.79. Huber Loss is a smart combination of the two. It behaves like MSE for small errors (providing a smooth gradient) and like MAE for large errors (providing robustness to outliers). It performed the best.

### 2.4.3 Comparison of Optimizers

The model was trained with three different optimizers:

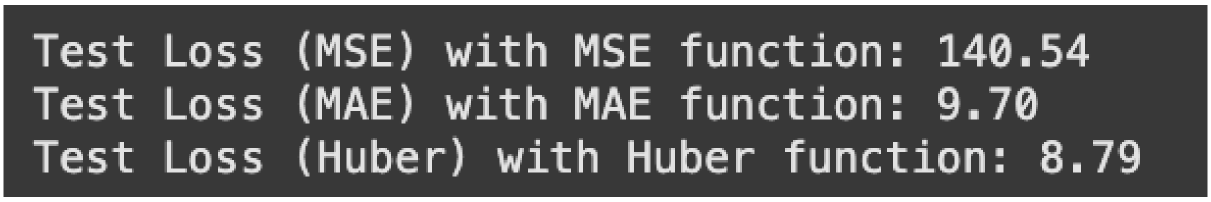
- **SGD:** Test MSE = 208.42
- **Adam:** Test MSE = 102.76
- **RMSprop:** Test MSE = 114.07

**Analysis:** The Adam optimizer achieved the best performance with the lowest error. Adaptive optimizers like Adam and RMSprop, which adjust the learning rate for each parameter individually, generally converge much faster and often find better solutions than standard SGD. The performance difference is clearly visible in the validation loss curves.



```
Test MSE with 20 epochs: 508.03
Test MSE with 50 epochs: 139.01
Test MSE with 100 epochs: 88.45
```

**Figure 15:** Numerical results of the experiments on epochs and loss functions.



```
Test Loss (MSE) with MSE function: 140.54
Test Loss (MAE) with MAE function: 9.70
Test Loss (Huber) with Huber function: 8.79
```

**Figure 16:** Numerical results of the optimizer comparison.

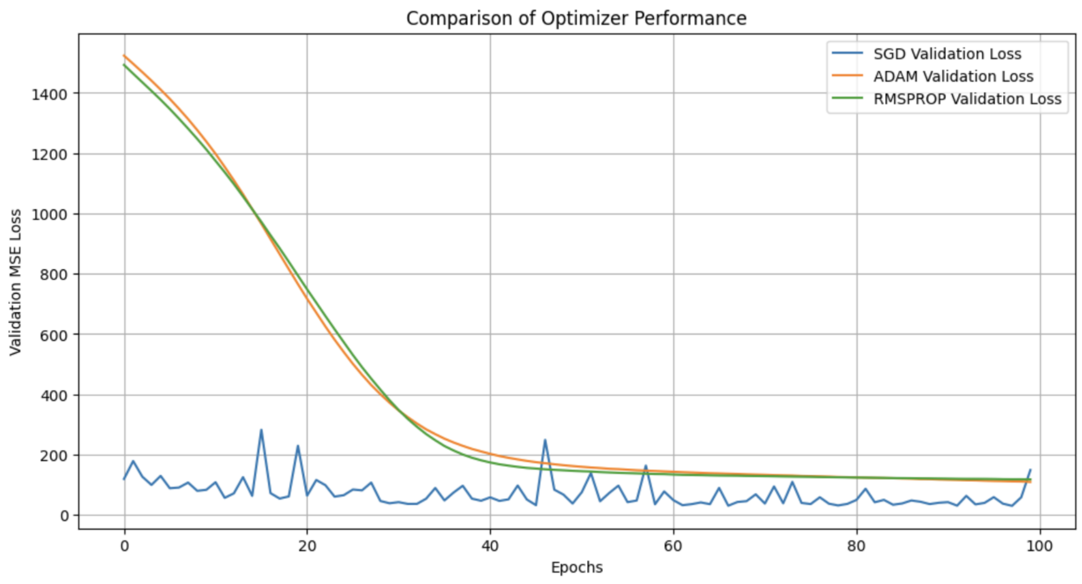


**Figure 17:** Comparison of validation loss curves for SGD, Adam, and RMSprop optimizers. Adam and RMSprop converge significantly faster and to a lower error than SGD.

## 2.5 Final Conclusion

- **Most Influential Feature:** ‘Cement‘ had the most significant impact on concrete strength, which is scientifically logical.
- **Best Model Settings:** Based on our experiments, the best settings were: 32 neurons, 100+ epochs (with early stopping), Huber Loss, and the Adam optimizer.
- **Primary Challenges:** The main challenges in this regression problem were selecting the right model architecture and hyperparameters, and dealing with the inherent noise in the data, as the relationship between the features and the target is not perfectly deterministic.

# 3 Question 3: Implementing Adaline for the IRIS Dataset

*Note: The student's report structure for this question is followed, integrating text and figures for a cohesive narrative.*

## 3.1 Introduction

This exercise aims to build a foundational understanding of one of the earliest neural network models: **Adaline (Adaptive Linear Neuron)**. We will implement the Adaline algorithm from scratch and train it to perform binary classification on a subset of the famous Iris dataset. The key focus will be on understanding its learning mechanism and the critical role of the learning rate hyperparameter on model convergence and performance.

## 3.2 Introduction to Adaline and Madaline

- **Adaline (Adaptive Linear Neuron):** Adaline is a single-layer neural network designed for linear classification and regression. Its key feature is the use of a linear activation function and a weight update rule based on the Gradient Descent learning algorithm. It attempts to minimize a cost function, typically the Sum of Squared Errors (SSE), between the network's continuous output and the target labels. This continuous output makes its learning process more stable than its predecessor, the Perceptron.

- **Madaline (Multiple Adaline):** This is an extension of Adaline and one of the first multilayer neural networks. A simple Madaline architecture consists of multiple Adaline neurons in a hidden layer, whose outputs are fed into a final output neuron (often implementing a logic function like AND or OR). Madaline is capable of solving more complex, non-linearly separable problems.

- **Difference from MLP:** The primary difference lies in the learning algorithm. In classic Madaline, the hidden layer weights are trained with the Adaline rule, while the output layer weights are often fixed. In contrast, a modern MLP uses the **Backpropagation** algorithm, which propagates the error from the output layer back through all layers, updating all network weights in a unified and integrated manner. Additionally, MLPs typically use non-linear activation functions (like Sigmoid or ReLU), which gives them the power to solve highly complex problems.

## 3.3 Data Preparation

1. **Dataset Loading and Subset Selection:** We load the Iris dataset from ‘sklearn‘. To create a binary classification problem, we keep only two classes, ‘Setosa‘ and ‘Versicolor‘, and discard the ‘Virginica‘ class. For ease of visualization, we use only two features: ‘petal length‘ and ‘petal width‘.

2. **Normalization and Splitting:** The feature values are normalized to the range [0, 1] to improve the performance of the gradient descent algorithm. The data is then split into a 70% training set (70 samples) and a 30% test set (30 samples). The class labels are mapped to -1 and 1.

## 3.4 Implementation and Training of the Adaline Model

The Adaline algorithm was implemented in Python. The model was then trained with three different learning rates to observe the effect on convergence.

- High Learning Rate: 0.02
- Medium Learning Rate: 0.005
- Low Learning Rate: 0.001

**Final Results after 10 Epochs:**

- LR = 0.02: Final Error = 10.78, Final Accuracy = 100%
- LR = 0.005: Final Error = 9.36, Final Accuracy = 100%
- LR = 0.001: Final Error = 29.56, Final Accuracy = 50%

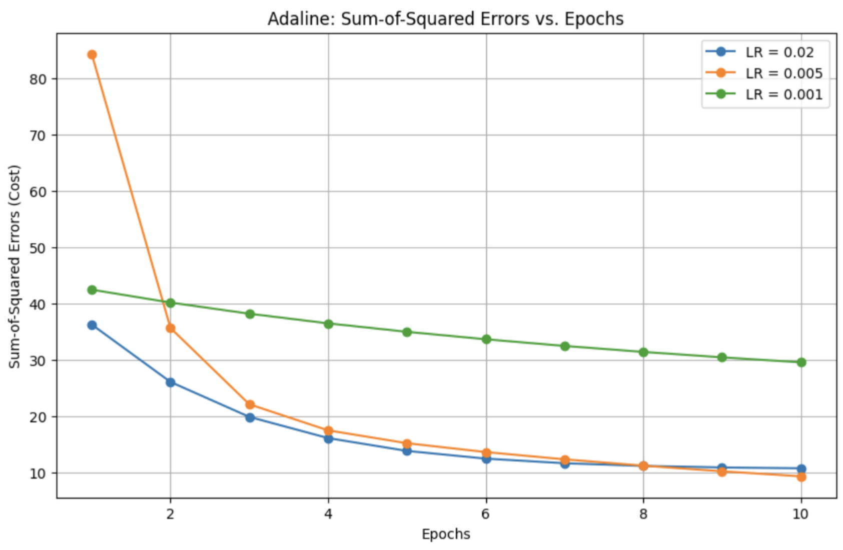


**Figure 18:** The Sum-of-Squared Errors (Cost) decreases over epochs for different learning rates. The higher learning rates (0.02 and 0.005) converge quickly, while the low learning rate (0.001) converges very slowly.

## 3.5 Display and Analysis of Results

### 3.5.1 Decision Boundary Visualization

The final separating line (decision boundary) was plotted for each of the three models on both the training and test data. As seen in the plots, the models with appropriate learning rates (0.02 and 0.005) found an optimal decision boundary that perfectly separates the two classes. The model with the low learning rate (0.001) failed to find a proper separating line.

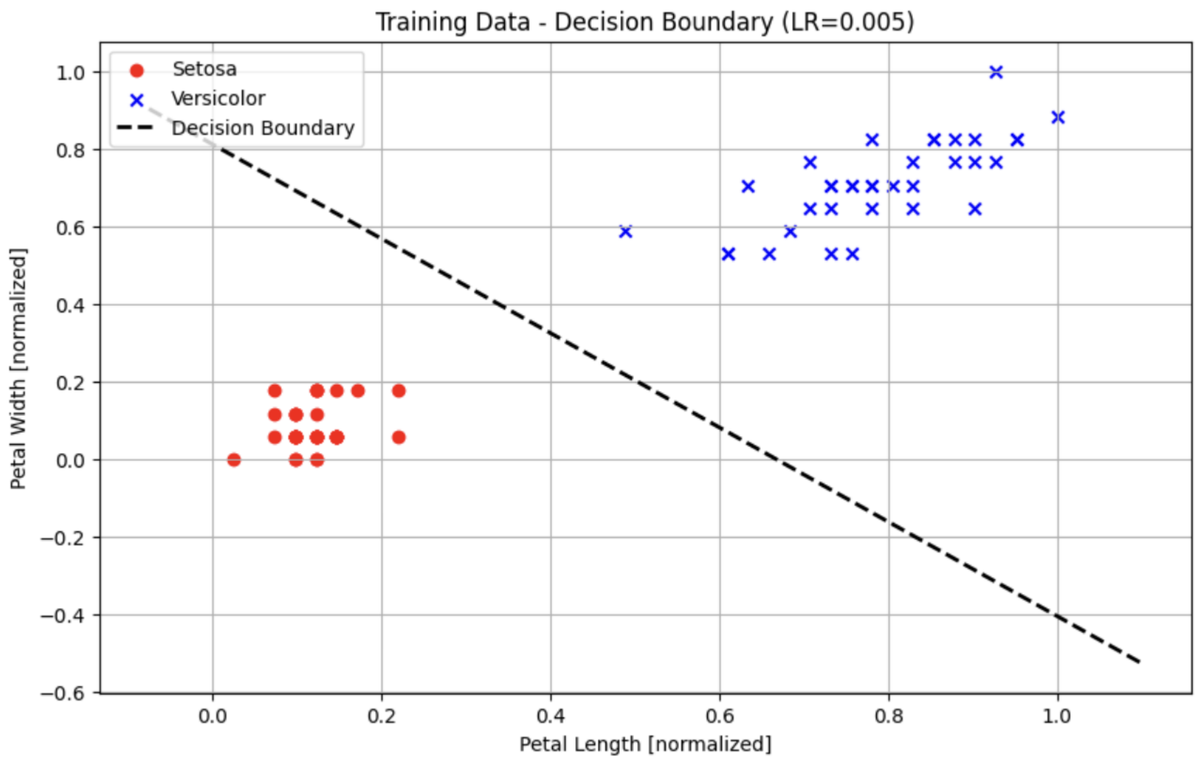


**Figure 19:** Decision boundaries on the training data for LR=0.02 (top) and LR=0.005 (bottom).

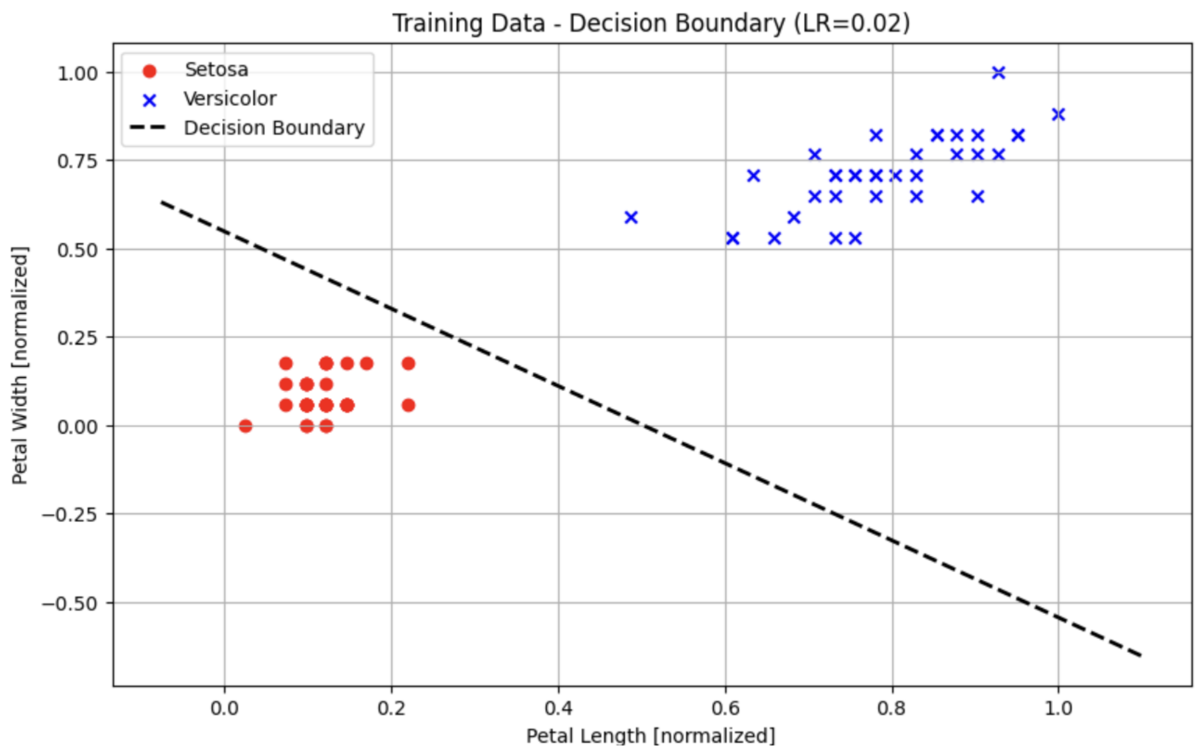


**Figure 20:** Decision boundaries for LR=0.001 on training data (top) and LR=0.02 on test data (bottom).

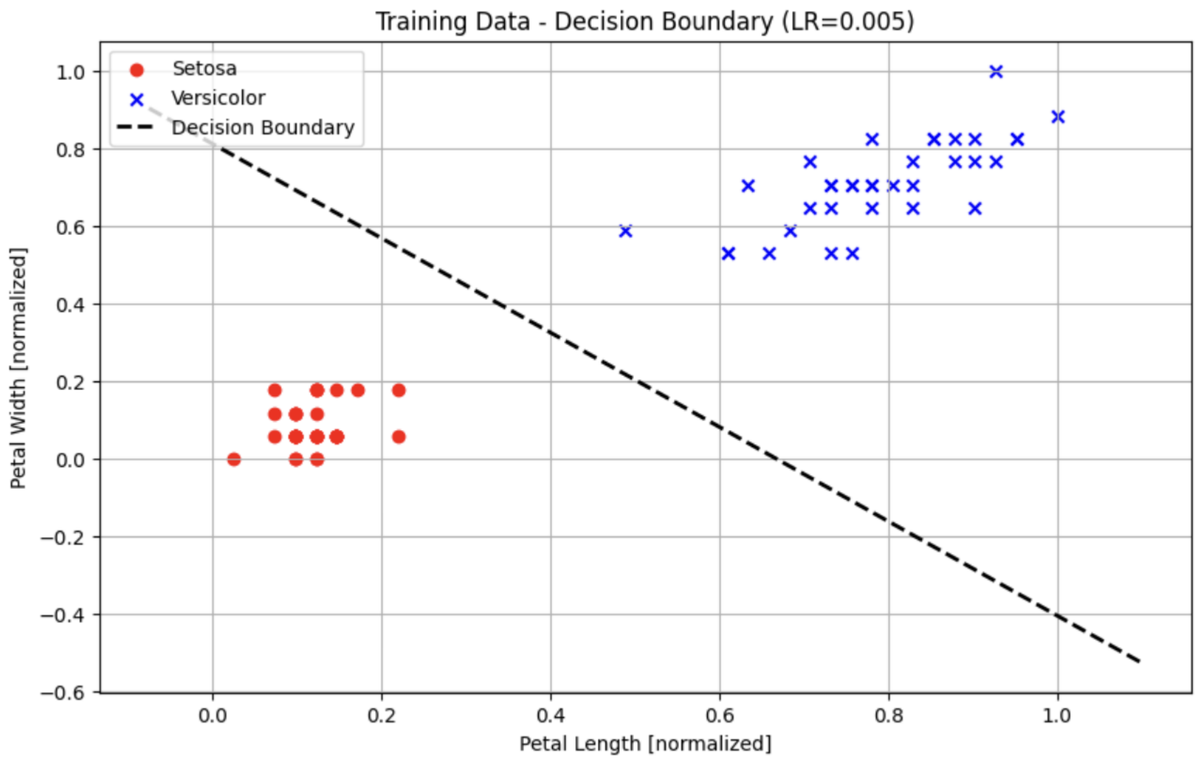


**Figure 21:** Decision boundaries on the test data for LR=0.005 (top) and LR=0.001 (bottom).

### 3.5.2 Result Analysis

- **Convergence:** The models with LR=0.02 and LR=0.005 converged successfully. The model with LR=0.001 did not converge within 10 epochs. The learning rate was too small, causing the weight updates to be minuscule and preventing the model from reaching the optimal solution in a reasonable time.
- **Effect of Linear Separability:** This subset of the Iris dataset is linearly separable, meaning a straight line can be drawn to perfectly separate the two classes. This is a critical property for the success of a linear classifier like Adaline. If the data were not linearly separable, Adaline would never be able to achieve 100% accuracy.
- **Effect of Learning Rate:** The learning rate had a decisive impact on performance. The high rate (0.02) was very effective for this problem. The medium rate (0.005) was also an excellent, "safe" choice. The low rate (0.001) was ineffective. This highlights that selecting an appropriate learning rate is one of the most crucial steps in training neural networks.

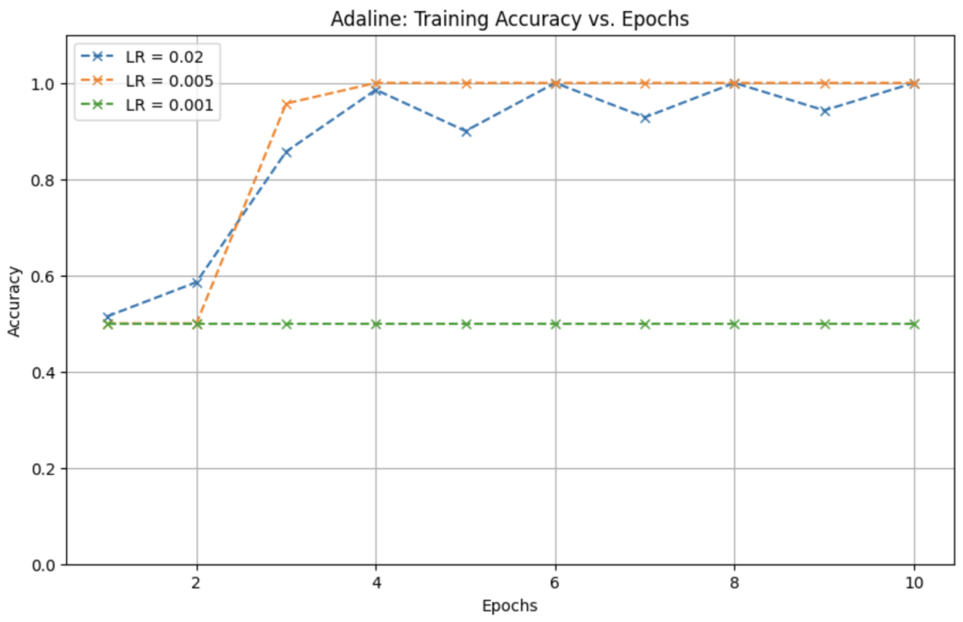


**Figure 22:** Training accuracy over epochs. The models with higher learning rates quickly achieve 100% accuracy, while the model with the lowest learning rate remains at 50% (random chance).

# 4 Question 4: Autoencoder and Classification with MNIST

*Note: The student's report combines the analysis and figures for this section. The following reflects that integrated structure.*

## 4.1 Introduction

In this project, we explore the concept of unsupervised representation learning using an **Autoencoder**. We will train an autoencoder to learn a compressed, meaningful representation (a latent space) of the MNIST handwritten digit dataset. Subsequently, we will use the encoder part of this model as a fixed feature extractor for a simple classification network. The primary goal is to investigate how the size of the latent space affects both image reconstruction quality and classification accuracy.

## 4.2 Data Downloading and Preprocessing

The MNIST dataset, consisting of 60,000 training and 10,000 test images of handwritten digits (0-9), was loaded using TensorFlow. The data was preprocessed as follows:

- **Normalization:** Pixel values were scaled from the [0, 255] range to [0, 1].
- **Flattening:** Each 28x28 pixel image was reshaped into a 784-dimensional vector to be used as input for our fully-connected (Dense) layers.

## 4.3 Model Design and Implementation

The process was divided into two main parts.

### 4.3.1 Part One: Training the Autoencoders

Two autoencoder models with identical architectures but different latent space dimensions were designed and trained for 20 epochs using Mean Squared Error (MSE) as the loss function.

- **Model AE-8 (8-dimensional latent space):**
  - Encoder: 784 -¿ 128 (ReLU) -¿ 8 (ReLU)
  - Decoder: 8 -¿ 128 (ReLU) -¿ 784 (Linear)
- **Model AE-4 (4-dimensional latent space):**
  - Encoder: 784 -¿ 128 (ReLU) -¿ 4 (ReLU)
  - Decoder: 4 -¿ 128 (ReLU) -¿ 784 (Linear)

### 4.3.2 Part Two: Training the Classifiers

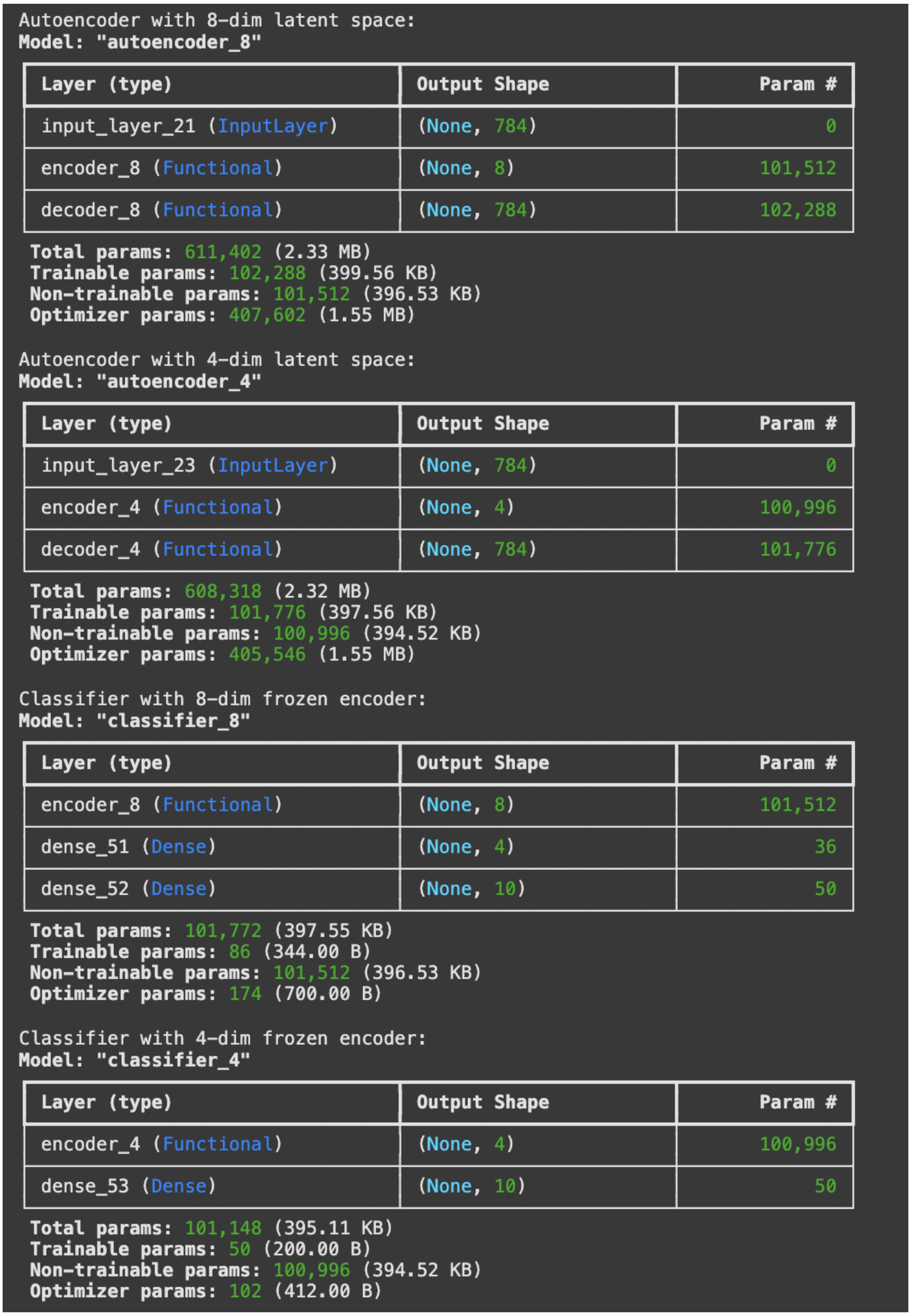
In this stage, the trained encoders from Part One were used as fixed feature extractors. A simple classification head was attached to the output of each **frozen** encoder. The resulting models were then trained on the labeled MNIST data.

- **Classifier-8 (using AE-8 encoder):**
  - Frozen Encoder Output: 8 dimensions
  - Classification Head: 8 -¿ 4 (ReLU) -¿ 10 (Softmax)
- **Classifier-4 (using AE-4 encoder):**
  - Frozen Encoder Output: 4 dimensions
  - Classification Head: 4 -¿ 10 (Softmax)

## 4.4 Results and Analysis

### 4.4.1 Parameter Counts

An interesting observation is the small number of trainable parameters in the classifier models. This demonstrates the efficiency of using a frozen encoder; the vast majority of the network's parameters are pre-trained and fixed.



```
Autoencoder with 8-dim latent space:
Model: "autoencoder_8"
┌──────────────────────────────┬───────────────────────┬──────────┐
│ Layer (type)                 │ Output Shape          │  Param # │
├──────────────────────────────┼───────────────────────┼──────────┤
│ input_layer_21 (InputLayer)  │ (None, 784)           │        0 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ encoder_8 (Functional)       │ (None, 8)             │  101,512 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ decoder_8 (Functional)       │ (None, 784)           │  102,288 │
└──────────────────────────────┴───────────────────────┴──────────┘
 Total params: 611,402 (2.33 MB)
 Trainable params: 102,288 (399.56 KB)
 Non-trainable params: 101,512 (396.53 KB)
 Optimizer params: 407,602 (1.55 MB)

Autoencoder with 4-dim latent space:
Model: "autoencoder_4"
┌──────────────────────────────┬───────────────────────┬──────────┐
│ Layer (type)                 │ Output Shape          │  Param # │
├──────────────────────────────┼───────────────────────┼──────────┤
│ input_layer_23 (InputLayer)  │ (None, 784)           │        0 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ encoder_4 (Functional)       │ (None, 4)             │  100,996 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ decoder_4 (Functional)       │ (None, 784)           │  101,776 │
└──────────────────────────────┴───────────────────────┴──────────┘
 Total params: 608,318 (2.32 MB)
 Trainable params: 101,776 (397.56 KB)
 Non-trainable params: 100,996 (394.52 KB)
 Optimizer params: 405,546 (1.55 MB)

Classifier with 8-dim frozen encoder:
Model: "classifier_8"
┌──────────────────────────────┬───────────────────────┬──────────┐
│ Layer (type)                 │ Output Shape          │  Param # │
├──────────────────────────────┼───────────────────────┼──────────┤
│ encoder_8 (Functional)       │ (None, 8)             │  101,512 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ dense_51 (Dense)             │ (None, 4)             │       36 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ dense_52 (Dense)             │ (None, 10)            │       50 │
└──────────────────────────────┴───────────────────────┴──────────┘
 Total params: 101,772 (397.55 KB)
 Trainable params: 86 (344.00 B)
 Non-trainable params: 101,512 (396.53 KB)
 Optimizer params: 174 (700.00 B)

Classifier with 4-dim frozen encoder:
Model: "classifier_4"
┌──────────────────────────────┬───────────────────────┬──────────┐
│ Layer (type)                 │ Output Shape          │  Param # │
├──────────────────────────────┼───────────────────────┼──────────┤
│ encoder_4 (Functional)       │ (None, 4)             │  100,996 │
├──────────────────────────────┼───────────────────────┼──────────┤
│ dense_53 (Dense)             │ (None, 10)            │       50 │
└──────────────────────────────┴───────────────────────┴──────────┘
 Total params: 101,148 (395.11 KB)
 Trainable params: 50 (200.00 B)
 Non-trainable params: 100,996 (394.52 KB)
 Optimizer params: 102 (412.00 B)
```

**Figure 23:** Model summaries showing the parameter counts for all four models.

### 4.4.2 Reconstruction Error Comparison

The model with the 8-dimensional latent space (AE-8) reconstructed the images significantly better than the 4-dimensional model (AE-4). The MSE loss curves show that the final reconstruction error for AE-8 ( 0.024) was much lower than for AE-4 ( 0.045). **Why?** A latent space of 8 dimensions has a higher capacity to retain the key information and details of the original image. Compressing the image down to just 4 dimensions forces the model to discard a large amount of useful information, resulting in blurry and inaccurate reconstructions.

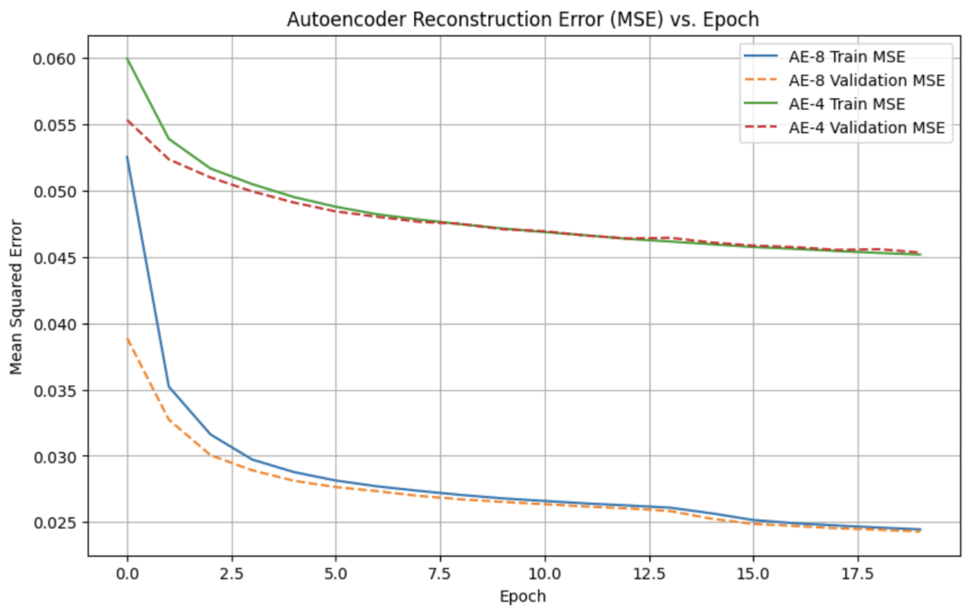


**Figure 24:** Autoencoder reconstruction error (MSE) during training. The 8-dim model (blue/orange) achieves a significantly lower error than the 4-dim model (green/red).

### 4.4.3 Classification Accuracy Comparison

The classifier attached to the 8-dim encoder performed vastly better, achieving a final test accuracy of approximately 82%. In contrast, the classifier with the 4-dim encoder was completely unsuccessful, with its accuracy hovering around 10-20%, not much better than random guessing. **Why?** The 8-dimensional latent space was able to preserve the distinguishing features between the different digits. The extreme compression to 4 dimensions destroyed these critical features, creating a useless representation for classification. This is a classic example of the trade-off between compression and information preservation.

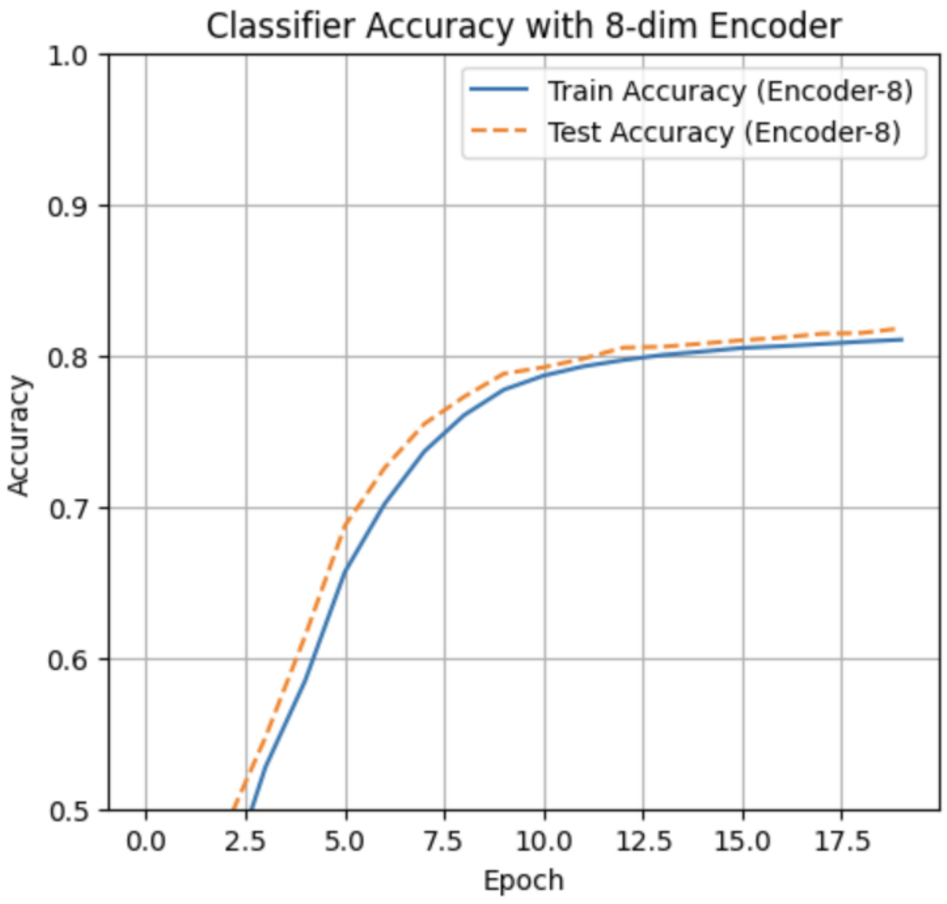


**Figure 25:** Classification accuracy for the model using the 8-dimensional encoder.

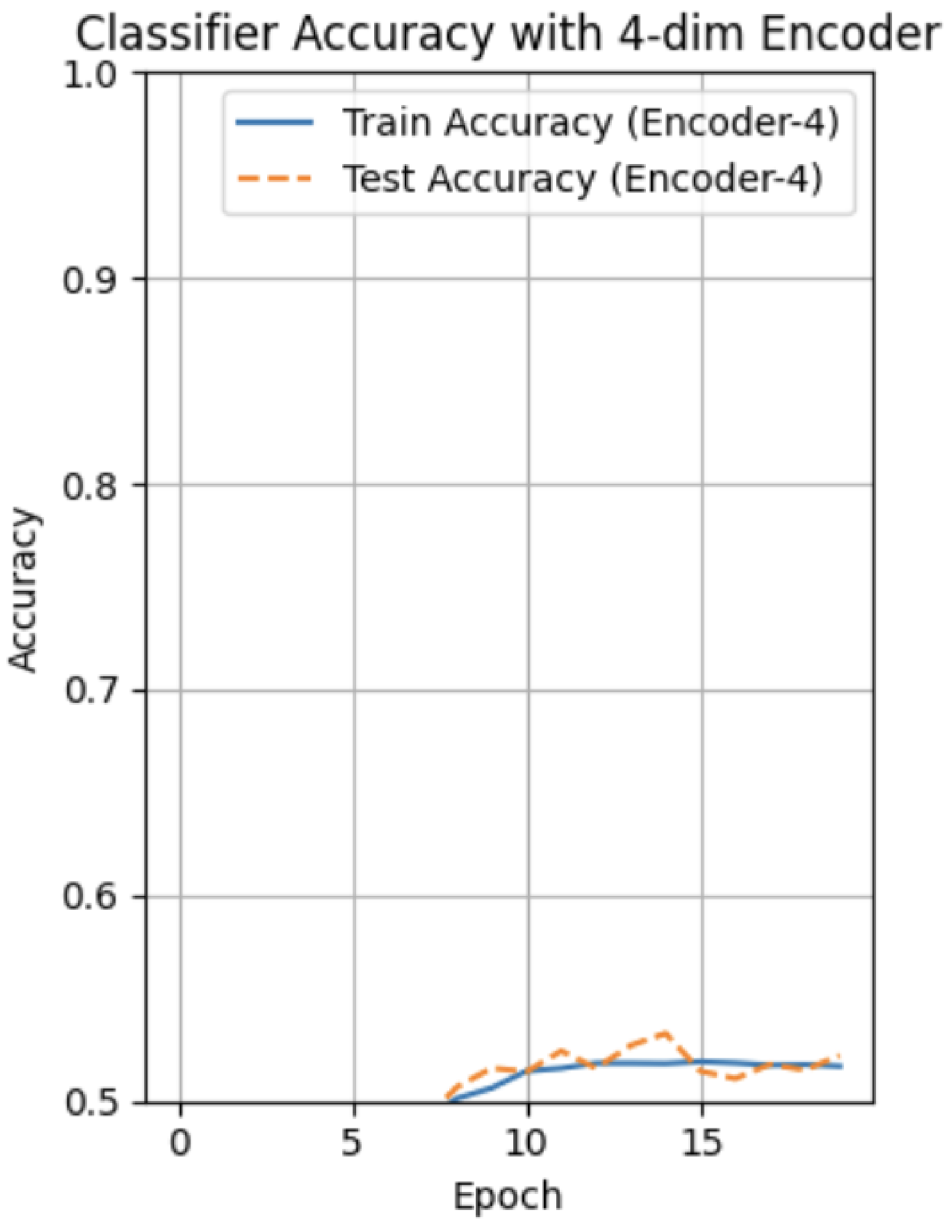
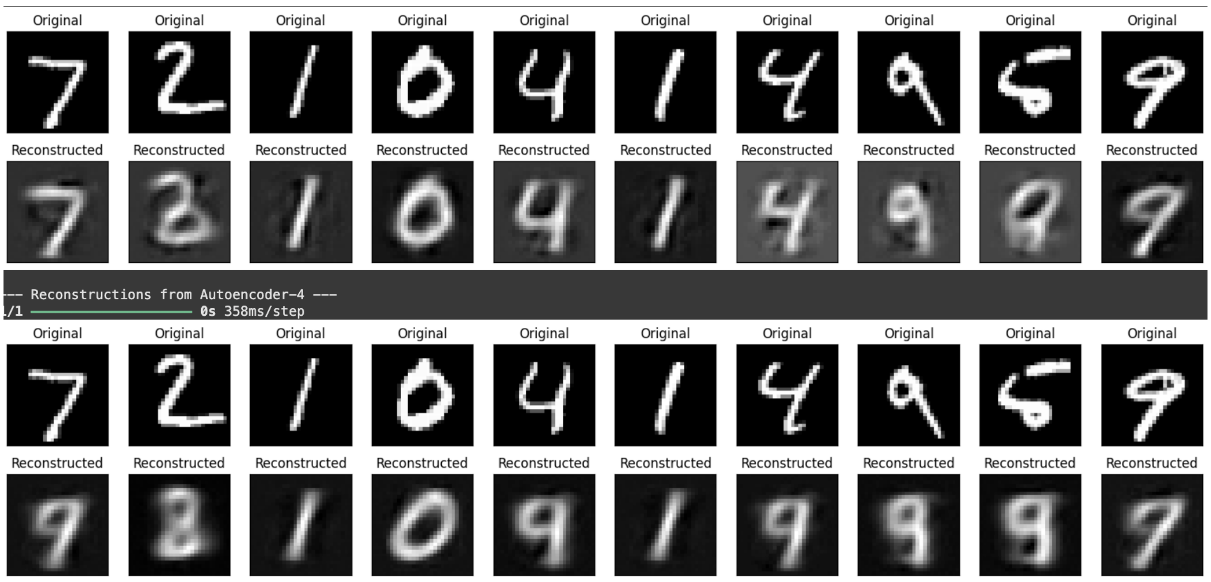


**Figure 26:** Classification accuracy for the model using the 4-dimensional encoder, showing a failure to learn.

### 4.4.4 Visual Comparison of Reconstructions

The reconstructed images confirm the quantitative results. Images reconstructed by AE-8 are clear and recognizable, while those from AE-4 are blurry and indistinct.



```
--- Reconstructions from Autoencoder-4 ---
/1 ━━━━━━━━━━━━━━━━━━━━ 0s 358ms/step
```

**Figure 27:** Visual comparison of original and reconstructed MNIST digits from the different autoencoder models.

### 4.4.5 Bonus: Implementing and Evaluating a Convolutional Autoencoder

The results from the simple, dense autoencoders demonstrated a clear proof-of-concept but also highlighted a significant architectural weakness. The final classification accuracy of 82% is far from state-of-the-art for MNIST. To address this, a more powerful **Convolutional Autoencoder (CAE)** was proposed and implemented. This section details the results of that improved model.

**Rationale for a Convolutional Architecture** The primary weakness of the initial models was their use of fully-connected (Dense) layers, which discard all spatial information by flattening the 28x28 images into 784-dimensional vectors. A CAE, by contrast, uses convolutional layers that are specifically designed to process grid-like data like images. They preserve spatial relationships, use parameter sharing for efficiency, and learn a hierarchical representation of features (from simple edges to complex shapes). This makes them vastly more suitable for image-based tasks.

**Final Results of the Improved Model** The implemented CAE, featuring a deeper architecture with convolutional layers, was trained for both reconstruction and subsequent classification. The performance leap was substantial and confirmed the hypothesis that a convolutional architecture is superior for this task.

- **Final Test Accuracy of Bonus Model: 95.16%**

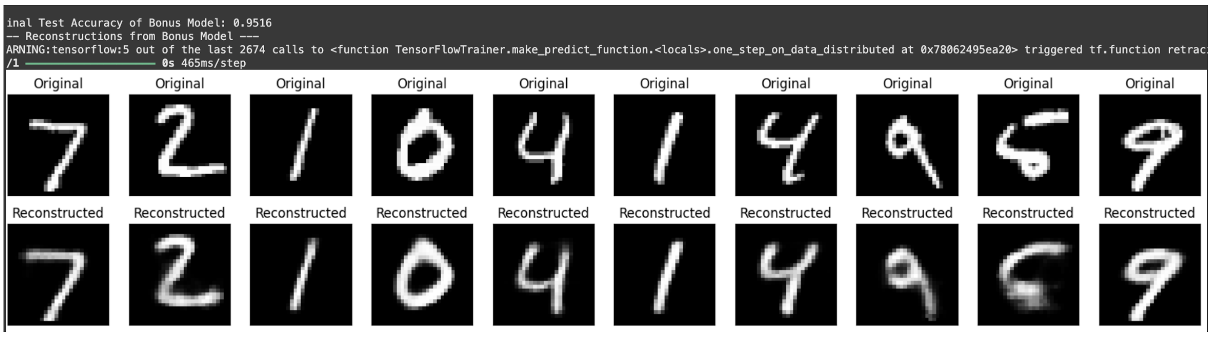
This result represents a massive improvement over the 82% accuracy of the best dense encoder. It moves the model from a basic proof-of-concept to a genuinely effective classifier.

**Analysis of Performance Improvement** The dramatic increase in accuracy can be attributed to the superior feature representation learned by the convolutional encoder.

1. **Richer Latent Space:** The convolutional layers extracted a much more meaningful and discriminative set of features. The resulting latent space effectively captured the essential ”digit-ness” of each image, making the classification task significantly easier for the final dense layers.

2. **Superior Reconstruction Quality:** As a direct consequence of better feature learning, the reconstruction quality of the CAE is visibly superior. Figure 28 shows that the reconstructed digits are sharp, clear, and almost indistinguishable from the originals. This contrasts sharply with the blurry outputs of the dense autoencoders, especially the 4-dimensional one. The high-fidelity reconstruction is a strong indicator that the encoder is successfully capturing the most important information from the input images.

3. **Efficiency and Effectiveness:** The CAE provides a clear demonstration of how choosing the right architectural inductive bias (i.e., using convolutions for images) leads to a model that is not only more accurate but also more efficient at learning useful representations.



```
inal Test Accuracy of Bonus Model: 0.9516
-- Reconstructions from Bonus Model ---
ARNING:tensorflow:5 out of the last 2674 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x78062495ea20> triggered tf.function retrac
/1 ━━━━━━━━━━━━━━━━━━━━ 0s 465ms/step
```

**Figure 28:** Reconstructions from the bonus Convolutional Autoencoder model. The visual quality of the reconstructed digits is exceptionally high, closely matching the originals. This demonstrates the superior ability of the convolutional architecture to learn and preserve key image features, which directly translates to the high classification accuracy of 95.16%.

In conclusion, the experiment with the Convolutional Autoencoder successfully validates the proposed path for improvement. By aligning the model architecture with the inherent spatial structure of the image data, we achieved a significant boost in performance for both the unsupervised reconstruction task and the supervised classification task.