University of Tehran

School of Electrical and Computer

Engineering

Neural Networks and Deep Learning

Assignment 1

Students:

Question	Name	Student ID
1	Sepehr Jamali	810101400
2	Kasra Ghorbani	810101489
3	Sepehr Jamali	810101400
4	Kasra Ghorbani	810101489

Contents

List of Figures **4**

List of Tables **7**

1 McCulloch–Pitts Neuron **8**

1–1 McCulloch–Pitts Neuron 8

1–2 Majority and Parity 10

2 Adaline and Madaline **14**

2–1 Adaline for breast cancer detection 14

2–1–1 From **SSE** to **LMS** 14

2–1–2 Data preparation and initial analysis 15

2–1–3 Implementing Adaline from scratch 16

2–1–4 Evaluation and displaying results 17

2–1–5 Baseline models 21

2–2 Madaline for Moons and Circles dataset 22

2–2–1 Implementing Madaline with Rule II 22

2–2–2 Data and training configuration 23

2–2–3 Evaluation and visualization 25

2–2–4 Studying hyperparameters 28

2–3 Results across different random seeds 29

3 MLP **30**

3–1 Getting Dataset and Preprocessing 30

3–2 Implementing Neural Network 33

3–3 Training Neural Network 33

3–4 Implementing and Training Improved Neural Network 36
3–5 Comperrsion 38

4 AutoEncoders **40**

4–1 Getting data and performing appropriate preprocessing 40
4–2 Implementing the AutoEncoder 41
4–3 Training the Network 42
4–4 Evaluating the network's performance 44
4–4–1 Encoding the data 44
4–4–2 Clustering with K-Means 45
4–4–3 Analyzing the clusters 45
4–4–4 Dimensionality reduction and t-SNE 46
4–4–5 Comparison with raw data 47
4–4–6 Evaluating clustering performance 48

List of Figures

1 McCulloch–Pitts neuron implementing the AND logic. 8

2 McCulloch–Pitts neuron implementing the OR logic. 9

3 McCulloch–Pitts neuron implementing the NOT logic. 9

4 3D representation of the XOR function with three binary inputs. Each corner of the cube corresponds to one possible input combination (x_0, x_1, x_2). Orange points represent outputs where $y = 1$ (odd number of 1's), and blue points represent outputs where $y = 0$ (even number of 1's). As seen, no single plane can separate the orange and blue points simultaneously, showing that XOR is not linearly separable. 11

5 Structure of XOR function neural network . The network consists of an input layer with three inputs (x_0, x_1, x_2), a hidden layer with 4 neurons and an output neuron. Weights (w_{ij}) and bias terms (b_i) are adjusted during training to form non-linear decision boundaries. 11

6 Neural network structure for the Majority function. The network has 5 input neurons $(x_0, \ldots, x_4)$, a hidden layer with 10 neurons (one for each 3-input combination), and 1 output neuron that computes the logical OR of the hidden neurons. Each hidden neuron detects whether its corresponding triple of inputs is all ones. 13

7 Confusion matrix for the Adaline classifier on the test set. 18

8 ROC curve for the Adaline classifier on the test set. 19

9 PCA-projected test data with Adaline decision boundary (dashed line). . . 20

10 Training MSE per epoch for different learning rates. 20

11 Training accuracy per epoch for different learning rates. 21

12 Training MSE for different Madaline configurations (Moons) 25

13 Training accuracies for different Madaline configurations (Moons) 25
14 Training MSE for different Madaline configurations (Circles) 25
15 Training accuracies for different Madaline configurations (Circles) 25
16 Madaline's decision boundary for moons 26
17 Madaline's decision boundary for circles 26
18 Confusion matrix for Madaline on Moons 27
19 ROC curve for Madaline on Moons 27
20 Confusion matrix for Madaline on Circles 27
21 ROC curve for Madaline on Circles 27
22 Preview of the California Housing dataset loaded in pandas. 30
23 Distribution of `total_bedrooms` after handling missing values using mean imputation. 32
24 First few rows of the dataset after applying one-hot encoding to `ocean_proximity`. The first category column is removed. 32
25 Reported parameters and architecture of the fully-connected neural network using `model.summary()`. 33
26 Loss curves for training, validation, and test datasets over epochs. The plot illustrates how the model's loss decreases and stabilizes during training. 34
27 Comparison of regression metrics (MAE, MSE, RMSE, R^2) for the training, validation, and test datasets. 36
28 Reported parameters and architecture of the improved fully-connected neural network using `model.summary()`. Batch normalization and dropout layers are included. 37
29 Loss curves for the improved neural network on training, validation, and test datasets. 38

30 Regression metrics (MAE, MSE, RMSE, R^2) for the improved neural network across training, validation, and test datasets. 38
31 Actual vs predicted values for the improved neural network. 39
32 residuals for the improved neural network. 40
33 Mean Squared Error (MSE) per epoch during AutoEncoder training. . . . 42
34 Original and reconstructed examples for the “T-shirt/top” class. 43
35 Original and reconstructed examples for the “Trouser” class. 43
36 Original and reconstructed examples for the “Pullover” class. 43
37 Original and reconstructed examples for the “Dress” class. 44
38 Original and reconstructed examples for the “Coat” class. 44
39 Confusion matrix for True Class vs. Cluster Assignments 45
40 t-SNE of Latent Space 46
41 t-SNE of Latent Space (Colored by K-Means Clusters) 46
42 Confusion Matrix (K-Means on raw data) 47
43 t-SNE of Raw MNIST Data (Colored by True Labels) 47
44 t-SNE of Raw MNIST Data (Colored by K-Means Clusters) 48

List of Tables

1 Truth table for the AND logic implemented by the McCulloch–Pitts neuron. 8

2 Truth table for the OR logic implemented by the McCulloch–Pitts neuron. 9

3 Truth table for the NOT logic implemented by the McCulloch–Pitts neuron. 10

4 Weights and biases between the input layer and hidden layer 12

5 Weights and bias from hidden layer to output layer. 12

6 Training data statistics after StandardScaler (full view) 16

7 Best-performing Adaline: hyperparameters 17

8 Evaluation metrics for the best-performing Adaline 17

9 Performance of different models 22

10 MADALINE network accuracy across different seeds and hyperparameter settings. 29

11 Features of the California Housing Dataset 30

1 McCulloch–Pitts Neuron

The McCulloch–Pitts neuron can be expressed as:

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where x_i are the inputs, w_i are the corresponding weights, b is the bias, and y is the output of the neuron.

1–1 McCulloch–Pitts Neuron

And logic:

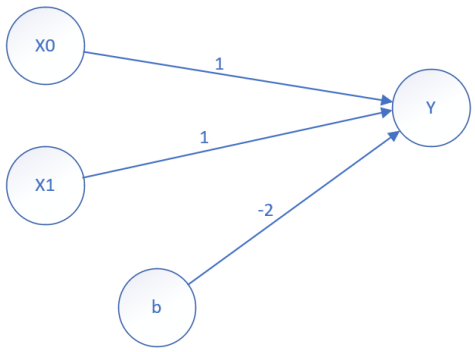

Figure 1: McCulloch–Pitts neuron implementing the AND logic.

The truth table for the AND operation is:

x_0	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: Truth table for the AND logic implemented by the McCulloch–Pitts neuron.

Or logic:

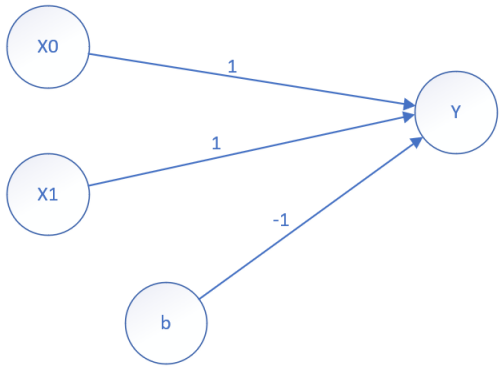

Figure 2: McCulloch–Pitts neuron implementing the OR logic.

The truth table for the OR operation is:

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: Truth table for the OR logic implemented by the McCulloch–Pitts neuron.

Not logic:

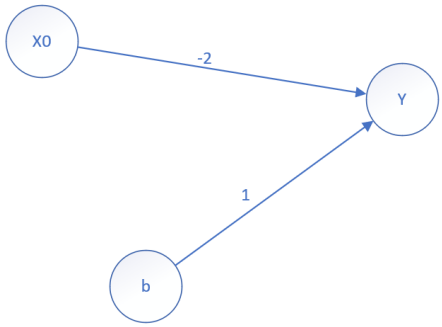

Figure 3: McCulloch–Pitts neuron implementing the NOT logic.

The truth table for the NOT operation is:

x_0	y
0	1
1	0

Table 3: Truth table for the NOT logic implemented by the McCulloch–Pitts neuron.

We can reason that the McCulloch–Pitts neural network is correct for the AND, OR and NOT operation because the weights and bias are chosen such that the neuron fires only when it should be. This is verified by the truth table , which shows that the output matches the expected logic for all possible input combinations. The neuron schematic further confirms the correct flow of weights and bias.

The McCulloch–Pitts neuron computes a weighted sum of inputs Then it applies a step function (threshold) to decide 0 or 1 Geometrically this is equivalent to drawing a line (in 2D) or hyperplane (in higher dimensions) that separates the input space into two regions: one where the neuron fires and one where it doesn't Because the neuron produces a single linear boundary, it can only separate points that are linearly separable.

1–2 Majority and Parity

The inputs x0,x1,x2 are vertices of a cube in three-dimensional space. Each vertex represents one possible binary combination. Vertices where the output is 1 (odd number of 1s) and Vertices where the output is 0 (even number of 1s) alternate across the cube. no single flat plane can separate all 1s from 0s.

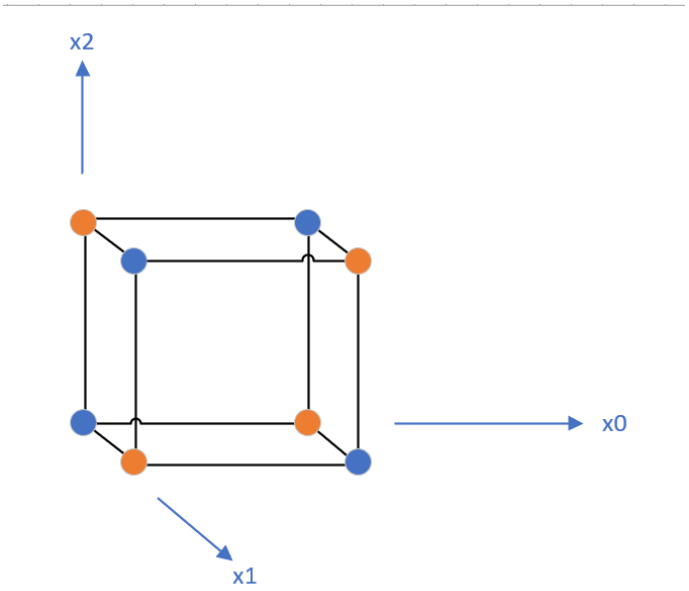

Figure 4: 3D representation of the XOR function with three binary inputs. Each corner of the cube corresponds to one possible input combination (x_0, x_1, x_2). Orange points represent outputs where $y = 1$ (odd number of 1's), and blue points represent outputs where $y = 0$ (even number of 1's). As seen, no single plane can separate the orange and blue points simultaneously, showing that XOR is not linearly separable.

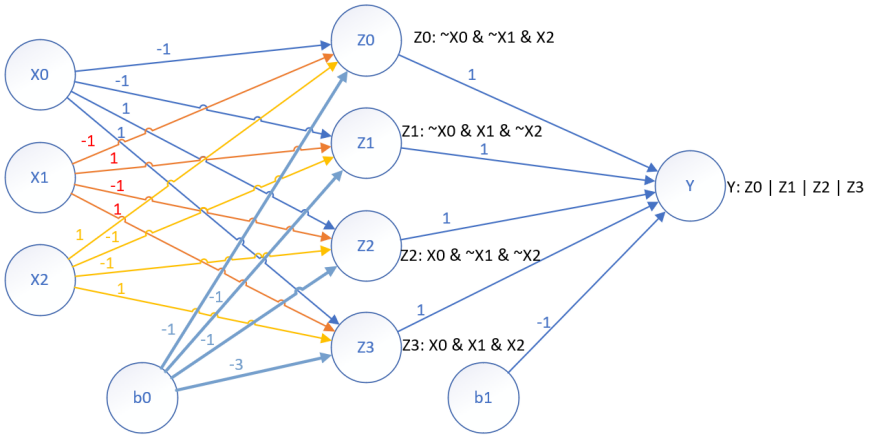

Figure 5: Structure of XOR function neural network . The network consists of an input layer with three inputs (x_0, x_1, x_2), a hidden layer with 4 neurons and an output neuron. Weights (w_{ij}) and bias terms (b_i) are adjusted during training to form non-linear decision boundaries.

Table 4: Weights and biases between the input layer and hidden layer

From / To	z_0	z_1	z_2	z_3
x_0	-1	-1	1	1
x_1	-1	1	-1	1
x_2	1	-1	-1	1
Bias b_0	-1	-1	-1	-3

Table 5: Weights and bias from hidden layer to output layer.

From Hidden Neuron	**Weight to Output** (y)
z_0	1
z_1	1
z_2	1
z_3	1
Bias b_1	-1

To implement a neural network for the **Majority function** with 5 inputs, the output should be 1 if at least 3 of the inputs are 1. Each hidden neuron detects one possible **triple** of inputs that are all 1. There are

$$\binom{5}{3} = 10$$

such triples, so we need **10 hidden neurons** in the hidden layer.

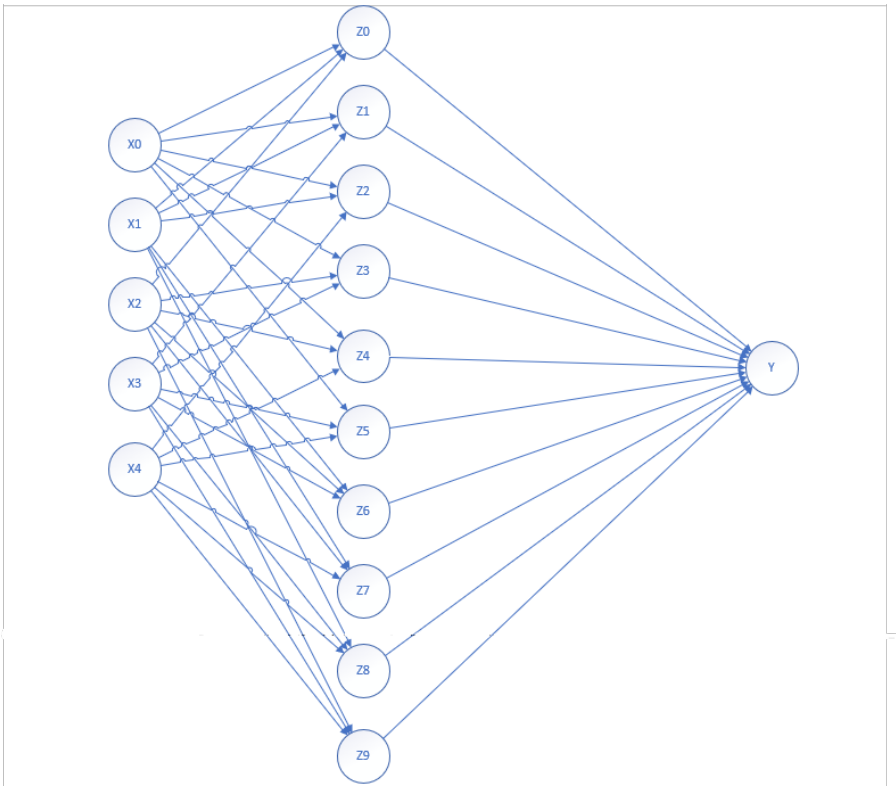

Figure 6: Neural network structure for the Majority function. The network has 5 input neurons ($x_0, \ldots, x_4$), a hidden layer with 10 neurons (one for each 3-input combination), and 1 output neuron that computes the logical OR of the hidden neurons. Each hidden neuron detects whether its corresponding triple of inputs is all ones.

2 Adaline and Madaline

2–1 Adaline for breast cancer detection

2–1–1 From SSE to LMS

For targets $t_n \in \{-1, +1\}$ and model $y_n = w^\top x_n$, define the mean squared error:

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} (t_n - w^\top x_n)^2.$$

Gradient:

$$\nabla_w E(w) = X^\top (Xw - t).$$

Normal equations (batch solution):

$$(X^\top X)w = X^\top t \quad \Rightarrow \quad w^\star = (X^\top X)^{-1} X^\top t.$$

Batch update rule:

$$w \leftarrow w + \eta \sum_{n=1}^{N} (t_n - w^\top x_n) x_n.$$

Online (LMS) update:

$$w \leftarrow w + \eta (t_n - w^\top x_n) x_n.$$

With L_2 regularization (ridge):

$$E_{\text{reg}}(w) = \tfrac{1}{2} \sum_{n} (t_n - w^\top x_n)^2 + \tfrac{\lambda}{2} \|w\|^2,$$

$$(X^\top X + \lambda I)w = X^\top t \quad \Rightarrow \quad w^\star = (X^\top X + \lambda I)^{-1} X^\top t.$$

Online update with L_2:

$$w \leftarrow (1 - \eta\lambda)w + \eta(t_n - w^\top x_n)x_n.$$

2–1–2 Data preparation and initial analysis

The Breast Cancer Wisconsin dataset from `sklearn.datasets` was used to test Adaline's performance. It contains 569 samples and 30 numeric features describing characteristics of cell nuclei (e.g., radius, texture, concavity). The data was divided into training and testing sets with a 70/30 ratio, maintaining class balance via `stratify=y`.

Before training, all features were standardized using **StandardScaler** to ensure zero mean and unit variance, as Adaline is sensitive to input scale. Basic exploratory analysis confirmed that both the training and test sets have similar class distributions, ensuring a representative split. Below the training data statistics are shown (after normalization) :

Table 6: Training data statistics after StandardScaler (full view)

Feature	**Mean**	**Std**	**Min**	**25%**	**50%**	**75%**	**Max**
mean radius	-3.34e-15	1.001	-1.773	-0.707	-0.243	0.461	3.871
mean texture	-6.74e-16	1.001	-2.229	-0.714	-0.092	0.606	4.703
mean perimeter	6.05e-15	1.001	-1.760	-0.714	-0.237	0.480	3.876
mean area	1.74e-16	1.001	-1.332	-0.676	-0.311	0.351	5.080
mean smoothness	-3.63e-15	1.001	-3.117	-0.752	-0.019	0.672	3.335
mean compactness	4.28e-16	1.001	-1.593	-0.766	-0.221	0.484	4.536
mean concavity	1.54e-16	1.001	-1.133	-0.762	-0.358	0.554	3.663
mean concave points	3.06e-16	1.001	-1.246	-0.735	-0.415	0.651	3.615
mean symmetry	-6.11e-16	1.001	-2.747	-0.678	-0.072	0.563	4.594
mean fractal dimension	2.50e-15	1.001	-1.806	-0.714	-0.205	0.457	4.903
radius error	8.21e-16	1.001	-1.008	-0.615	-0.301	0.260	8.599
texture error	-1.71e-15	1.001	-1.567	-0.705	-0.155	0.487	4.946
perimeter error	2.39e-16	1.001	-0.987	-0.606	-0.279	0.204	9.020
area error	-2.23e-18	1.001	-0.691	-0.468	-0.331	0.096	10.352
smoothness error	5.36e-17	1.001	-1.405	-0.622	-0.255	0.368	7.616
compactness error	-6.98e-16	1.001	-1.340	-0.702	-0.279	0.409	6.350
concavity error	2.99e-16	1.001	-1.183	-0.623	-0.209	0.424	10.146
concave points error	-2.23e-18	1.001	-1.931	-0.709	-0.132	0.494	4.729
symmetry error	2.68e-16	1.001	-1.321	-0.648	-0.219	0.326	6.948
fractal dimension error	1.48e-15	1.001	-1.230	-0.634	-0.231	0.354	7.857
worst radius	1.17e-15	1.001	-1.540	-0.690	-0.268	0.524	4.049
worst texture	4.82e-16	1.001	-2.241	-0.746	-0.056	0.667	3.570
worst perimeter	-9.48e-16	1.001	-1.541	-0.699	-0.275	0.529	4.238
worst area	-6.07e-16	1.001	-1.132	-0.643	-0.345	0.353	5.854
worst smoothness	-3.43e-15	1.001	-2.229	-0.709	-0.077	0.613	3.984
worst compactness	-7.48e-16	1.001	-1.402	-0.676	-0.252	0.520	4.351
worst concavity	8.28e-16	1.001	-1.322	-0.752	-0.212	0.544	4.790
worst concave points	3.01e-16	1.001	-1.725	-0.757	-0.229	0.701	2.645
worst symmetry	-9.91e-16	1.001	-2.180	-0.641	-0.110	0.475	6.149
worst fractal dimension	-1.24e-15	1.001	-1.607	-0.692	-0.255	0.444	5.000

2–1–3 Implementing Adaline from scratch

Adaline was implemented entirely from scratch in Python, without using high-level machine learning libraries. The implementation includes:

- Random initialization of weights and bias (using numpy $seed = 1$ for reproducibil-

ity).

- Iterative weight updates according to the LMS rule + L_2 penaltly.
- Mean Squared Error (MSE) monitoring for each epoch.
- Classification output threshold at 0 to map linear outputs to binary classes.

The model was trained with three different learning rates ($\eta = 0.001, 0.005, 0.02$) for up to 50 epochs. During each epoch, both the MSE and classification accuracy were computed to evaluate the convergence behavior.

2–1–4 Evaluation and displaying results

After training multiple Adaline models with different hyperparameter settings, the best-performing network had the following configuration:

Table 7: Best-performing Adaline: hyperparameters

Parameter	**Value**
Learning rate	0.005
L_2 penalty	0.001
Epochs	50

The corresponding evaluation metrics on the test set are reported in Table 8.

Table 8: Evaluation metrics for the best-performing Adaline

Metric	**Value**
Accuracy	0.9591
Precision	0.9545
Recall	0.9813
F1 score	0.9677

These results indicate strong and balanced classification performance: high precision shows that positive predictions are usually correct, while high recall indicates most pos-

itive cases are detected. The F1 score close to 0.97 confirms a good trade-off between precision and recall.

Confusion matrix. Figure 7 shows the confusion matrix for the best model on the test set. The matrix highlights that most errors are concentrated in a small number of misclassified samples, consistent with the high precision and recall values.

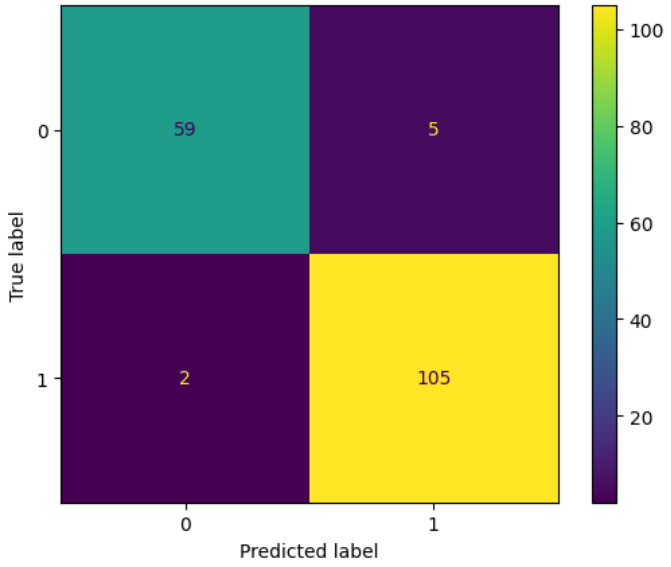

Figure 7: Confusion matrix for the Adaline classifier on the test set.

ROC curve and AUC. The ROC curve (Figure 8) and its AUC quantify discrimination ability across thresholds. A high AUC indicates the model reliably ranks positive instances above negative ones.

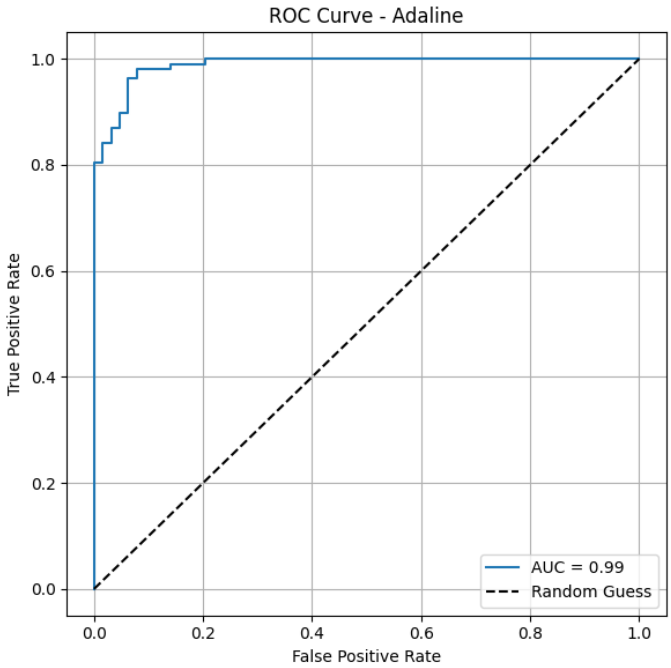

Figure 8: ROC curve for the Adaline classifier on the test set.

Decision boundary (PCA projection). To visualize the learned linear boundary, the test data were projected to two dimensions using PCA and the Adaline decision boundary was plotted in that space (Figure 9). The linear separator aligns well with the class clusters in the reduced space, illustrating that the problem is largely linearly separable after scaling.

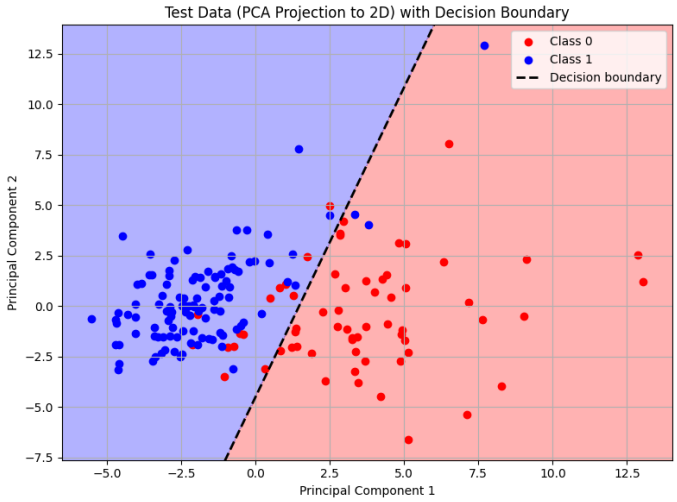

Figure 9: PCA-projected test data with Adaline decision boundary (dashed line).

Training dynamics. Figures 10 and 11 display the training MSE and accuracy per epoch for the evaluated learning rates.

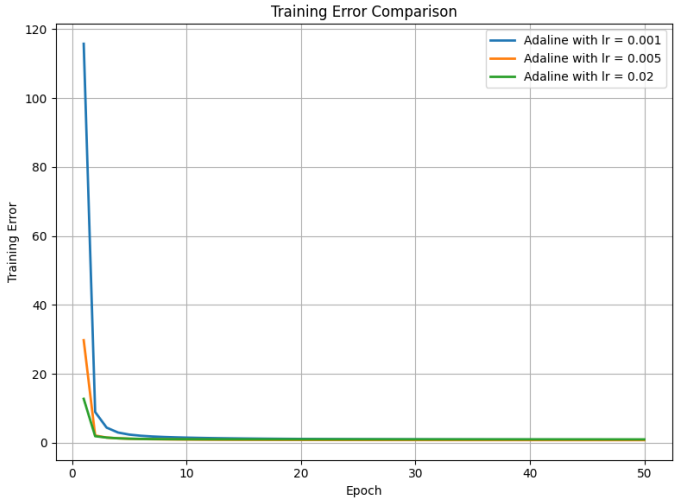

Figure 10: Training MSE per epoch for different learning rates.

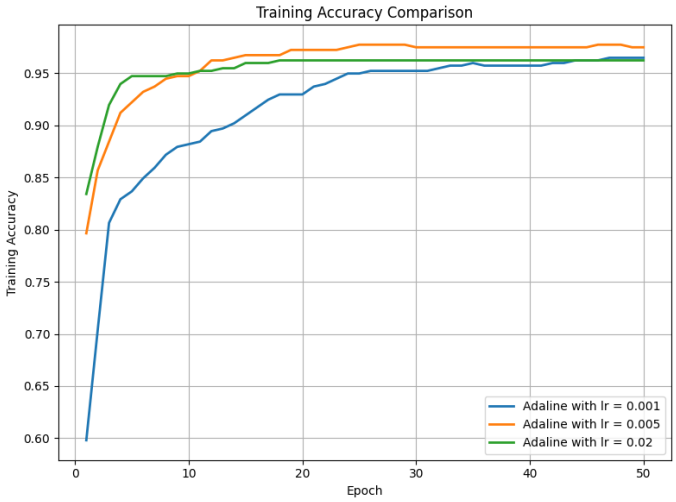

Figure 11: Training accuracy per epoch for different learning rates.

2–1–5 Baseline models

To benchmark Adaline's performance, two standard classifiers were trained on the same dataset:

- **Logistic Regression:** A linear probabilistic model optimized using the log-loss function.
- **Linear Discriminant Analysis (LDA):** A generative model assuming Gaussian class-conditional distributions.

Both baseline models achieved high classification accuracy (above 90%), similar to or slightly lower than the Adaline implementation. This confirms that the manually implemented Adaline model correctly approximates linear decision boundaries for this dataset, although more advanced optimization (e.g., batch updates or adaptive learning rates) could further improve stability and performance.

Discussion:

Adaline's performance depends strongly on the linear separability of the dataset and the

Table 9: Performance of different models

Model	**Accuracy**
Adaline	98.84%
Logistic Regression	93.56%
LDA	95.32%

choice of learning rate and regularization parameters. Since Adaline minimizes the mean squared error rather than the log-loss (as in logistic regression), it performs comparably on linearly separable problems but can be less robust when the data contain outliers or are not perfectly separable. In contrast, Logistic Regression optimizes a probabilistic objective and typically yields better-calibrated probabilities and more stable convergence in noisy or overlapping class scenarios. LDA, being a generative model, performs best when its Gaussian and equal-covariance assumptions hold; otherwise, it may underperform compared to Adaline. Overall, Adaline may outperform these baselines when the decision boundary is approximately linear and the noise level is low, while its performance tends to degrade in highly nonlinear settings.

2–2 Madaline for Moons and Circles dataset

2–2–1 Implementing Madaline with Rule II

Madaline Rule II is a supervised learning algorithm for two-layer networks made of Adaline units. For each input pattern, the network output is computed and compared with the target. If the output is correct, no weights are changed. If incorrect, the algorithm temporarily flips the outputs of one or more hidden Adalines and selects the change that most reduces the overall error. This process repeats for all training samples until convergence.

Algorithm 1 Madaline training pseudo code

Input: X, y, learning rate η, epochs E, L2 regularization λ, number of hidden neurons n_h

Output: MSE list, Accuracy list

Initialize $\mathbf{W} \sim \mathcal{N}(0,1)$ of shape $(n_h, n_{features})$
Initialize $\mathbf{b} \sim \mathcal{N}(0,1)$ of shape $(n_h,)$
Initialize $\mathbf{w}_{out} \sim \mathcal{N}(0,1)$ of shape $(n_h,)$
Initialize $b_{out} \sim \mathcal{N}(0,1)$
Initialize empty lists MSE_list and Acc_list
For each epoch ep in 1 to E **Do**
    $total\_error \leftarrow 0$
    $y\_preds \leftarrow$ zero vector of length $n\_samples$
    For each sample $(\mathbf{x}_i, t_i)$ in (X, y) **Do**
        $z_{in} \leftarrow \mathbf{W} \cdot \mathbf{x}_i + \mathbf{b}$
        $z \leftarrow activation(z_{in})$
        $y_{in} \leftarrow \mathbf{w}_{out} \cdot z + b_{out}$
        $\hat{y}_i \leftarrow 1$ if $y_{in} \geq 0$ else -1
        Store $\hat{y}_i$ in y_preds
        If $\hat{y}_i \neq t_i$ **Then**
            $dz \leftarrow 1 - z^2$
            $\delta_{out} \leftarrow t_i - y_{in}$
            $\delta_{hidden} \leftarrow \delta_{out} \cdot \mathbf{w}_{out} \cdot dz$
            $\mathbf{w}_{out} \leftarrow \mathbf{w}_{out} + \eta\,(\delta_{out} z - \lambda \mathbf{w}_{out})$
            $b_{out} \leftarrow b_{out} + \eta\,\delta_{out}$
            $\mathbf{W} \leftarrow \mathbf{W} + \eta\,\left(\delta_{hidden} \mathbf{x}_i^T - \lambda \mathbf{W}\right)$
            $\mathbf{b} \leftarrow \mathbf{b} + \eta\,\delta_{hidden}$
        End If
        $total\_error \leftarrow total\_error + 0.5(t_i - y_{in})^2$
    End For
    $MSE \leftarrow total\_error / n\_samples$
    $Accuracy \leftarrow$ mean$(y_preds == y)$
    Append MSE to MSE_list and $Accuracy$ to Acc_list
    Print "Epoch ep: MSE = MSE, Accuracy = $Accuracy \times 100\%$"
End For
Return MSE_list, Acc_list =0

2–2–2 Data and training configuration

Two synthetic two-dimensional datasets were used to evaluate the Madaline implementation:

- **Moons:** `sklearn.datasets.make_moons` with 600 samples and noise 0.25.
- **Circles:** `sklearn.datasets.make_circles` with 600 samples and noise 0.10.

For each dataset the data were split into train and test sets (70% / 30%) with stratification to preserve class balance. Features were standardized using `StandardScaler` prior to training.

Model hyperparameters explored in the experiments include:

- hidden layer size m $(3, 5)$,
- learning rate η (used value: 0.001),
- number of epochs (100 for Moons, 100 for Circles),
- L_2 penalty λ (0 and 0.001).

Representative configurations used in the experiments:

- **Moons dataset:**
  1. Config 1: $\eta = 0.001$, $m = 3$, $\text{epochs} = 100$, $\lambda = 0.001$
  2. Config 2: $\eta = 0.001$, $m = 3$, $\lambda = 0$
  3. Config 3: $\eta = 0.001$, $m = 5$, $\lambda = 0.001$
  4. Config 4: $\eta = 0.001$, $m = 5$, $\lambda = 0$
- **Circles dataset:**
  1. Config 1: $\eta = 0.001$, $m = 3$, $\text{epochs} = 100$, $\lambda = 0.001$
  2. Config 2: $\eta = 0.001$, $m = 3$, $\lambda = 0$
  3. Config 3: $\eta = 0.001$, $m = 5$, $\lambda = 0.001$

4. Config 4: $\eta = 0.001$, $m = 5$, $\lambda = 0$

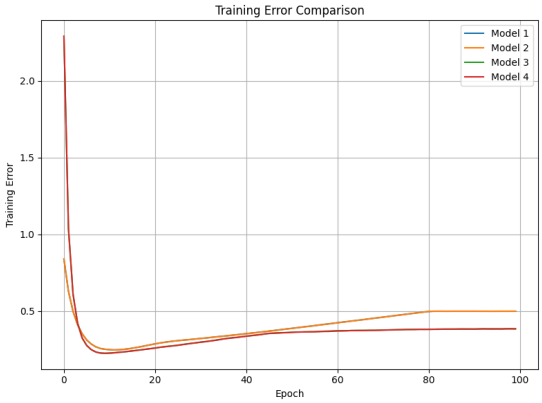

Figure 12: Training MSE for different Madaline configurations (Moons)

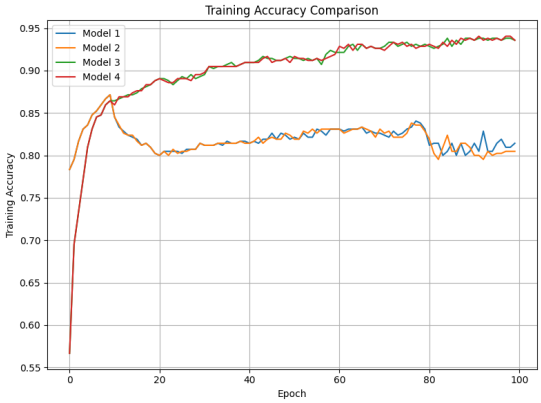

Figure 13: Training accuracies for different Madaline configurations (Moons)

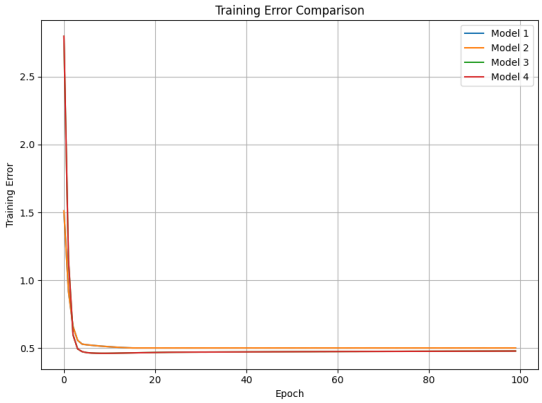

Figure 14: Training MSE for different Madaline configurations (Circles)

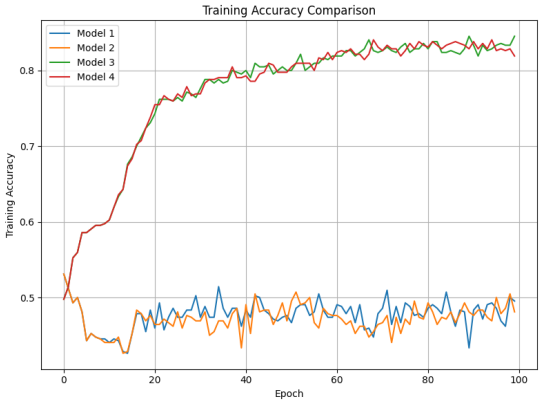

Figure 15: Training accuracies for different Madaline configurations (Circles)

2–2–3 Evaluation and visualization

For the moons dataset the best performing configuration was :

Config 3: $\eta = 0.001$, $m = 5$, $\lambda = 0.001$ and for the circles dataset it was Config 3: $\eta = 0.001$, $m = 5$, $\lambda = 0.001$. below we will display further results from these models

1. **Decision boundaries:** the classifier was evaluated on a dense grid and a contour plot was produced so the learned partition of the input space can be inspected.

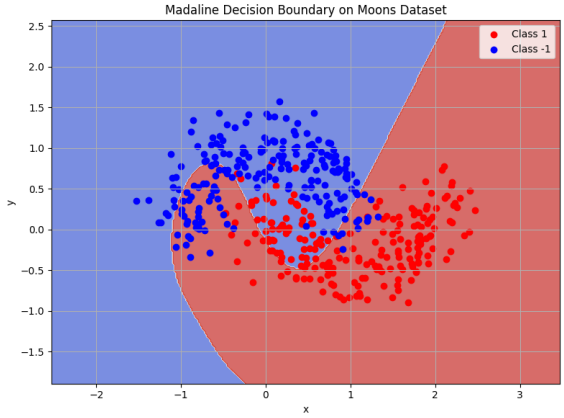

Figure 16: Madaline's decision boundary for moons

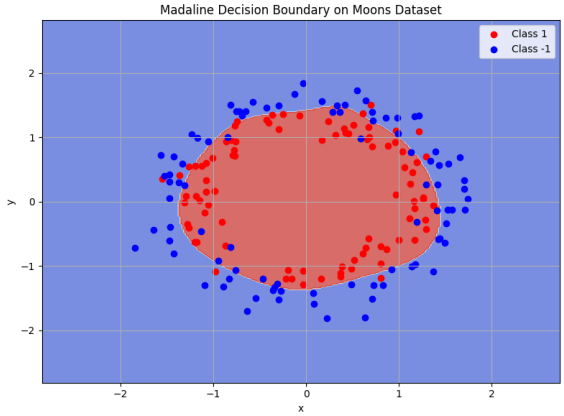

Figure 17: Madaline's decision boundary for circles

2. **ROC and confusion matrix figures:**

   Typical figures generated include:

   - Decision boundary plots for the best model on Moons and Circles datasets,
   - Confusion matrices and ROC curves with reported AUC values.

   For Moons :

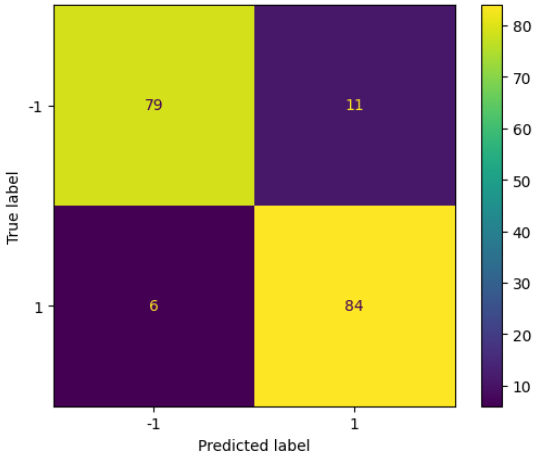

Figure 18: Confusion matrix for Madaline on Moons

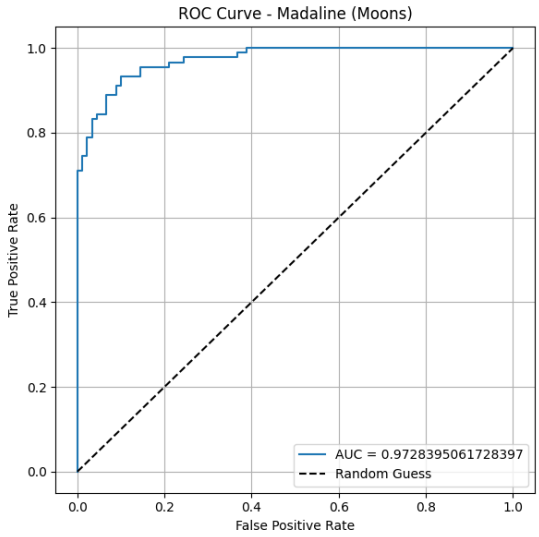

Figure 19: ROC curve for Madaline on Moons

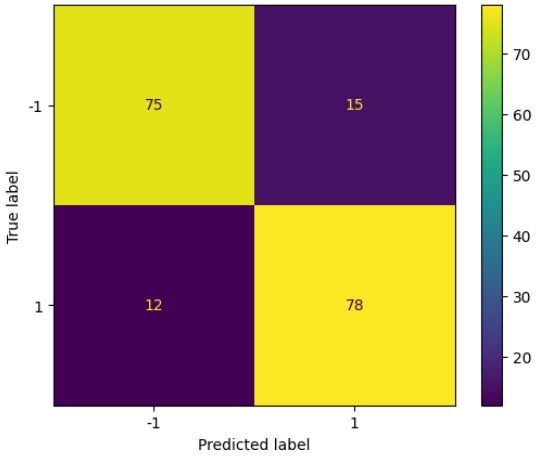

Figure 20: Confusion matrix for Madaline on Circles

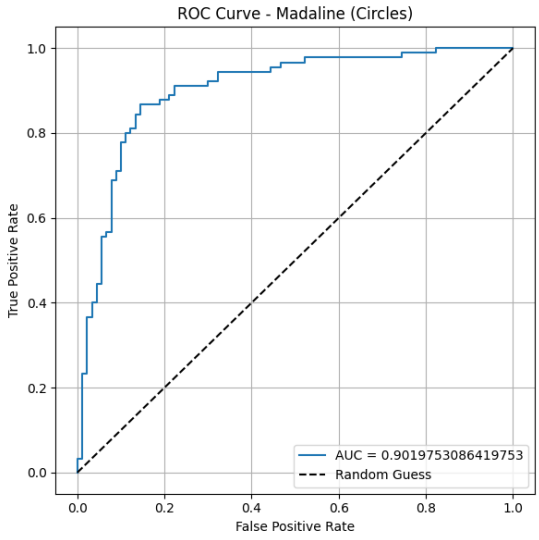

Figure 21: ROC curve for Madaline on Circles

In a MADALINE network, the final decision boundary emerges from the combination of multiple linear *hypersurfaces* produced by individual Adaline units in the hidden layer. Each Adaline computes a linear decision function which defines a hyperplane in the input

space that separates the feature space into two half-spaces. The outputs of these hidden units are then combined—typically through nonlinear activations or threshold logic—by the output layer to form a composite decision boundary. Geometrically, this means that the final classification surface is not a single hyperplane, but rather a piecewise-linear boundary constructed by the intersection and union of several such hyperplanes. As more hidden units are added, the network gains the ability to approximate increasingly complex, nonlinear regions in the input space by aligning and combining these hypersurfaces in different orientations.

2–2–4 Studying hyperparameters

From the experimental configurations the following qualitative trends were observed and should be reported:

- **Hidden layer size (m):** Increasing m generally improves expressivity and helps the network learn more complex decision boundaries (important for the *Circles* dataset). However, overly large hidden layers can increase training time and risk overfitting if not regularized.

- **Regularization (L_2):** Adding a small L_2 penalty ($\lambda > 0$) stabilizes updates and often improves generalization on the test set, particularly when the hidden layer is large or training runs for many epochs.

- **Learning rate (η):** A small learning rate ($\eta = 0.001$ in the experiments) produced stable learning and smooth decline in MSE. Larger learning rates can accelerate convergence but may cause oscillations or divergence in the online corrective scheme.

- **Epochs and convergence:** More epochs help when the model capacity is high

(larger m), but diminishing returns are observed once training error stabilizes. Monitoring training curves (MSE and accuracy) is essential to choose an appropriate stopping point.

- **Overfitting and Underfitting:** Overfitting occurs when the model learns the training data too precisely, capturing noise and irregularities instead of generalizable patterns. This typically happens when the hidden layer is too large or training continues for too many epochs. Underfitting, on the other hand, occurs when the model is too simple (e.g., small m or excessive regularization) to capture the true complexity of the data. The optimal decision boundary lies between these two extremes—complex enough to model the structure of the data but simple enough to generalize well to unseen samples.

2–3 Results across different random seeds

The table below reports the mean and standard deviation of the Accuracy score across 3 runs using seeds $0, 1, 2$, for both the `circles` and `moons` datasets under different hyperparameter settings.

Table 10: MADALINE network accuracy across different seeds and hyperparameter settings.

Dataset	**Model**	**Neurons**	L_2	**Accuracy Mean**	**Accuracy Std**
circles	1	3	0.001	0.5222	0.0111
circles	2	3	0.000	0.5056	0.0147
circles	3	5	0.001	0.8593	0.0210
circles	4	5	0.000	0.8574	0.0179
moons	1	3	0.001	0.8000	0.0242
moons	2	3	0.000	0.7833	0.0873
moons	3	5	0.001	0.9185	0.0225
moons	4	5	0.000	0.9222	0.0192

3 MLP

3–1 Getting Dataset and Preprocessing

First let's take a look at our dataset

Feature	Type	Description
longitude	Numerical	East–west coordinate of the block group. More negative → farther west (closer to the Pacific Ocean).
latitude	Numerical	North–south coordinate. Larger values → farther north in California.
housing_median_age	Numerical	Median age of houses in the block group (in years).
total_rooms	Numerical	Total number of rooms across all houses in the block group.
total_bedrooms	Numerical	Total number of bedrooms across all houses in the block group. May contain missing values.
population	Numerical	Total number of people living in the block group.
households	Numerical	Total number of households (occupied housing units).
median_income	Numerical	Median household income in the block group (in tens of thousands of dollars).
median_house_value	Numerical	Median house value for the block group (target variable, in USD).
ocean_proximity	Categorical	Distance from the ocean; possible values: NEAR BAY, ¡1H OCEAN, INLAND, NEAR OCEAN, ISLAND.

Table 11: Features of the California Housing Dataset

We load the dataset using pandas.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Figure 22: Preview of the California Housing dataset loaded in pandas.

Our first step in preprocessing is removing outliers using the **Interquartile Range**

(IQR).

The IQR is a measure of statistical dispersion, representing the range between the first quartile (Q1, 25th percentile) and the third quartile (Q3, 75th percentile) of the data:

$$\text{IQR} = Q_3 - Q_1$$

Outliers are defined as data points below $Q_1 - 1.5 \times \text{IQR}$ or above $Q_3 + 1.5 \times \text{IQR}$:

$$\text{Lower Bound} = Q_1 - 1.5 \times \text{IQR}$$

$$\text{Upper Bound} = Q_3 + 1.5 \times \text{IQR}$$

Any values outside these bounds are considered outliers and can be removed from the dataset. By applying this method to our dataset, we removed **1,071 data points** that were considered outliers.

Our next step in preprocessing is handling missing values. Our approach is to replace missing values with the **mean** of the respective column. Since the number of missing values is low (˜200), this simple technique is sufficient and does not significantly affect the dataset.

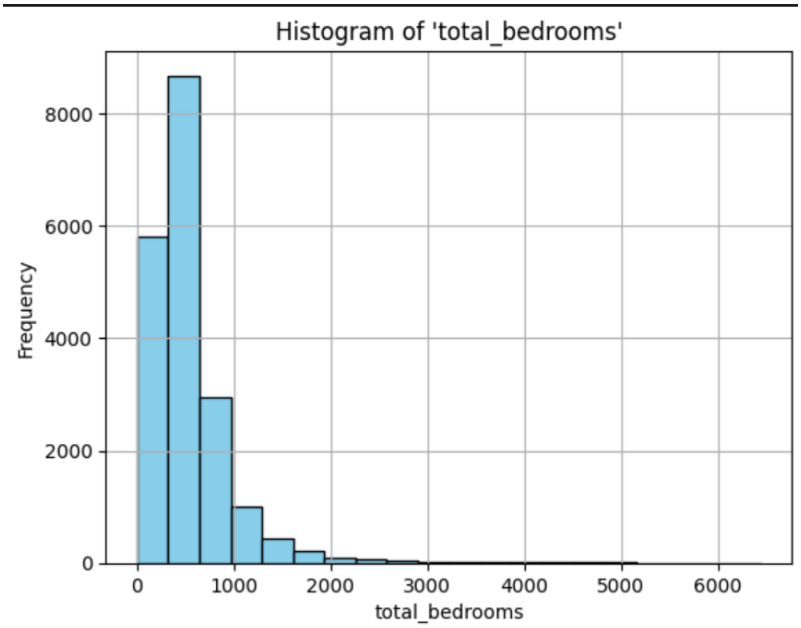

Figure 23: Distribution of `total_bedrooms` after handling missing values using mean imputation.

Our final step in preprocessing is handling categorical features, specifically `ocean_proximity` in this dataset. We will use **One-Hot Encoding** to convert the categorical values into numerical columns. This prevents introducing ordinal correlations between the values. Since this is a regression problem, we can remove the first column after encoding to reduce unnecessary processing (avoiding the dummy variable trap).

ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
False	False	True	False
False	False	True	False
False	False	True	False
False	False	True	False
False	False	True	False

Figure 24: First few rows of the dataset after applying one-hot encoding to `ocean_proximity`. The first category column is removed.

3–2 Implementing Neural Network

To implement a fully-connected (dense) neural network, we use **TensorFlow** and its `Dense` layers. We can report the parameters of the network using the `.build()` and `.summary()` functions, which provide an overview of the model architecture, number of neurons, and trainable parameters.

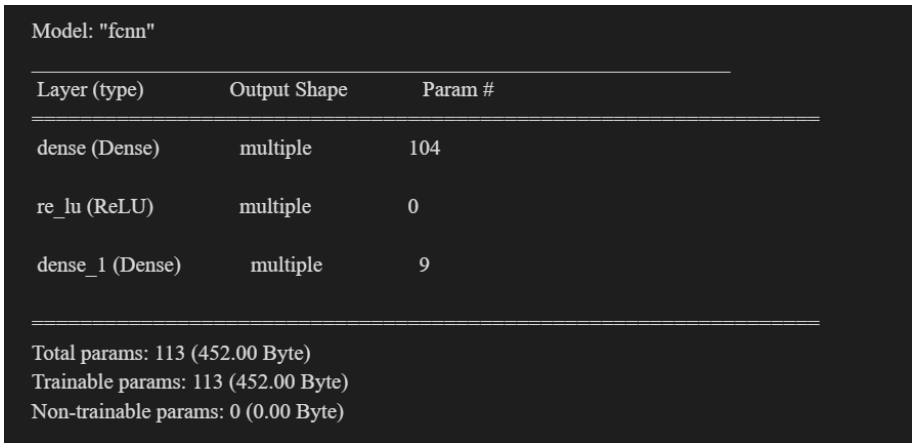

Figure 25: Reported parameters and architecture of the fully-connected neural network using `model.summary()`.

parameters are classified into **trainable** and **non-trainable** parameters. Trainable parameters are the ones that the network learns during training, typically including the weights and biases of each layer. These parameters are updated during backpropagation to minimize the loss function.

Non-trainable parameters, on the other hand, are part of the model but remain constant during training. Examples include fixed parameters in layers like BatchNormalization or pre-trained weights in transfer learning that we will use in the improved Neural network.

3–3 Training Neural Network

For training, we first split the dataset into **training**, **validation**, and **test** sets according to the ratio specified in the problem. After splitting, we convert the data into `NumPy`

arrays.

Next, we create batches of data using TensorFlow's `tf.data.Dataset`. Batching allows the network to process multiple samples at once, which improves computational efficiency and stabilizes gradient updates. We also shuffle the training data to ensure that each batch contains a random selection of samples, which helps in better generalization of the network.

We then iterate over the training batches for multiple epochs, updating the model parameters via backpropagation and monitoring the performance on the validation set.

After training, we plot the loss curves for the **training**, **validation**, and **test** datasets. These plots show how the model's performance evolves over epochs and help us visualize convergence, overfitting, or underfitting.

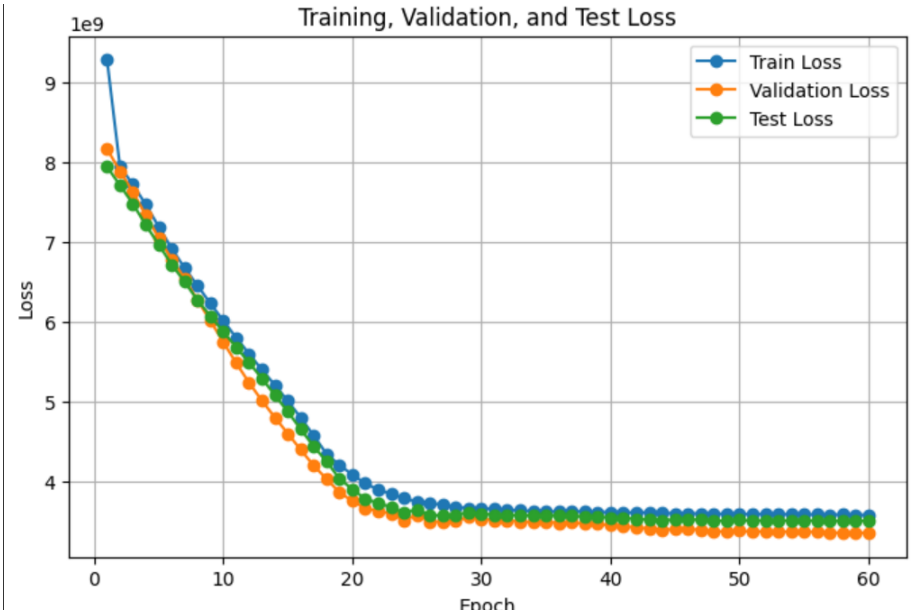

Figure 26: Loss curves for training, validation, and test datasets over epochs. The plot illustrates how the model's loss decreases and stabilizes during training.

After training the model, we evaluate its performance using common regression metrics: **Mean Absolute Error (MAE)**, **Mean Squared Error (MSE)**, **Root Mean Squared Error (RMSE)**, and **R-squared** (R^2).

1. Mean Absolute Error (MAE): MAE measures the average absolute difference between the predicted and actual values.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

2. Mean Squared Error (MSE): MSE calculates the average squared difference between predicted and actual values. It penalizes larger errors more heavily.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

3. Root Mean Squared Error (RMSE): RMSE is the square root of MSE and gives the error in the same units as the target variable.

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

4. R-squared (R^2): R^2 measures the proportion of variance in the target variable that is explained by the model. Values closer to 1 indicate better fit.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where $\bar{y}$ is the mean of the actual values.

These metrics provide complementary insights: MAE and RMSE measure absolute prediction errors, MSE emphasizes larger errors, and R^2 quantifies overall explained variance.

Dataset	MSE	RMSE	MAE	R^2
Train	[3544396500.0]	[59534.836]	[45410.168]	[0.60934454]
Validation	[3364587500.0]	[58005.066]	[45255.28]	[0.64668906]
Test	[3506974000.0]	[59219.71]	[45388.418]	[0.6089583]

Figure 27: Comparison of regression metrics (MAE, MSE, RMSE, R^2) for the training, validation, and test datasets.

3–4 Implementing and Training Improved Neural Network

For the improved neural network, we increase the number of layers to 5. The layers have the following number of neurons: 64, 32, 16, 8, and 1 (output layer).

We use `kernel_initializer='he_normal'` for each layer. The He normal initializer, sets the initial weights of the network according to a normal distribution with variance scaled by the number of input units to the layer. This method helps prevent vanishing or exploding gradients in deep networks and allows for faster and more stable convergence during training. In addition to increasing the number of layers, we apply **Batch Normalization** and **Dropout** in the improved neural network:

Batch Normalization: Batch normalization normalizes the inputs to each layer so that they have a mean of zero and a standard deviation of one. This helps stabilize and accelerate training.

Dropout: Dropout is a regularization technique where a fraction of neurons in a layer is randomly dropped during each training iteration. This prevents neurons from co-adapting too much, reducing overfitting. In our network we use decreasing dropout rates because we will get better results compared to static dropout rate.

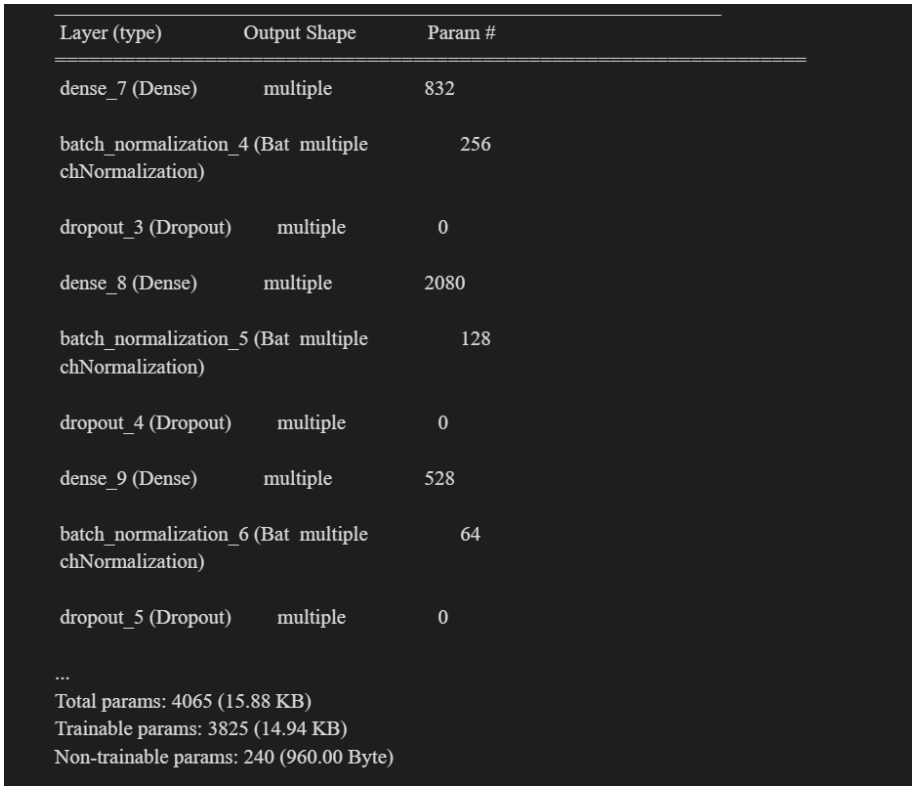

Figure 28: Reported parameters and architecture of the improved fully-connected neural network using `model.summary()`. Batch normalization and dropout layers are included.

Before training the improved neural network, we normalize our datasets. This ensures that all features contribute equally to the learning process and prevents features with larger numerical ranges from dominating the gradients during training.

When used together with **Batch Normalization** batch normalization further standardizes the inputs to each layer during training and stabilizing learning. normalizing the input data provides a good starting point, while batch normalization ensures stable and efficient training throughout the network.

After that, we split our data into batches and train the network as before, visualizing the loss plots for the training, validation, and test datasets.

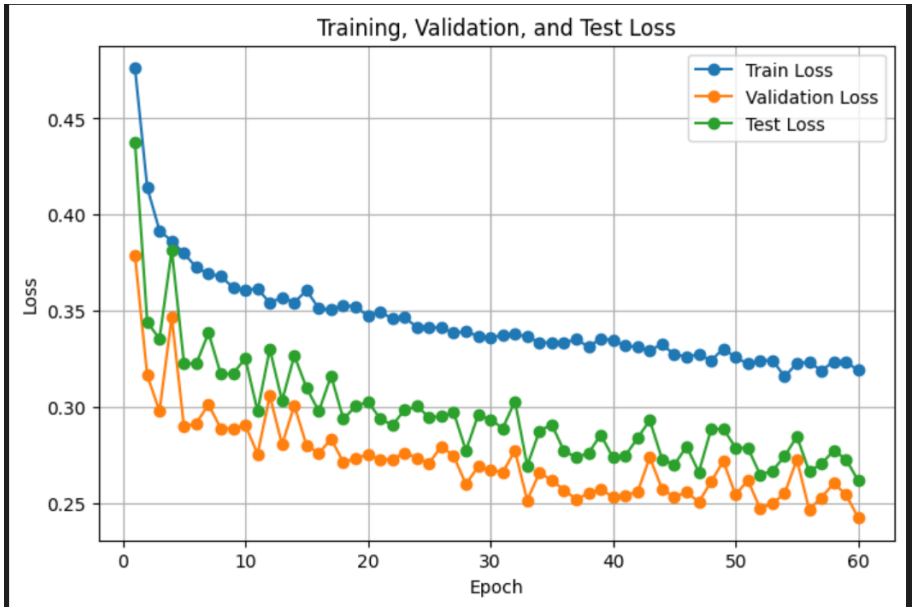

Figure 29: Loss curves for the improved neural network on training, validation, and test datasets.

Now we evaluate the results using the previously mentioned metrics (MAE, MSE, RMSE, and R^2). We can clearly see that the improved neural network achieves a better R^2 score.

Dataset	MSE	RMSE	MAE	R^2
Train	[0.25805414]	[0.5079903]	[0.35173258]	[0.7419449]
Validation	[0.24360327]	[0.49356183]	[0.3461919]	[0.76790965]
Test	[0.26245004]	[0.51229876]	[0.35570148]	[0.7344872]

Figure 30: Regression metrics (MAE, MSE, RMSE, R^2) for the improved neural network across training, validation, and test datasets.

3–5 Comperrsion

The hyperparameters are the same for both the original and improved fully-connected neural networks. By examining the loss plots, we can see that the loss of the first network stabilizes after approximately 25 epochs. In contrast, the improved network continues to

change until the last epoch, indicating that it is still learning and has not fully converged as quickly as the original network. This behavior can be attributed to the increased depth, batch normalization, and dropout in the improved architecture, which allow it to capture more complex patterns but may require more training iterations to stabilize.

R^2 is considered the best indicator of model performance. From our results, we can see that the improved neural network achieves approximately 10% higher R^2 compared to the original network, indicating a better fit to the data and improved predictive performance.

Actual vs Predicted Plot: This plot shows the predicted values of the target variable against the true values. Ideally, the points should lie close to the diagonal line $y = x$, indicating that predictions match the actual values well.

Residual Plot: The residuals are calculated as the difference between the actual and predicted values (Residual $= y - x$). Plotting residuals helps detect patterns or biases in predictions. Ideally, residuals should be randomly scattered around zero.

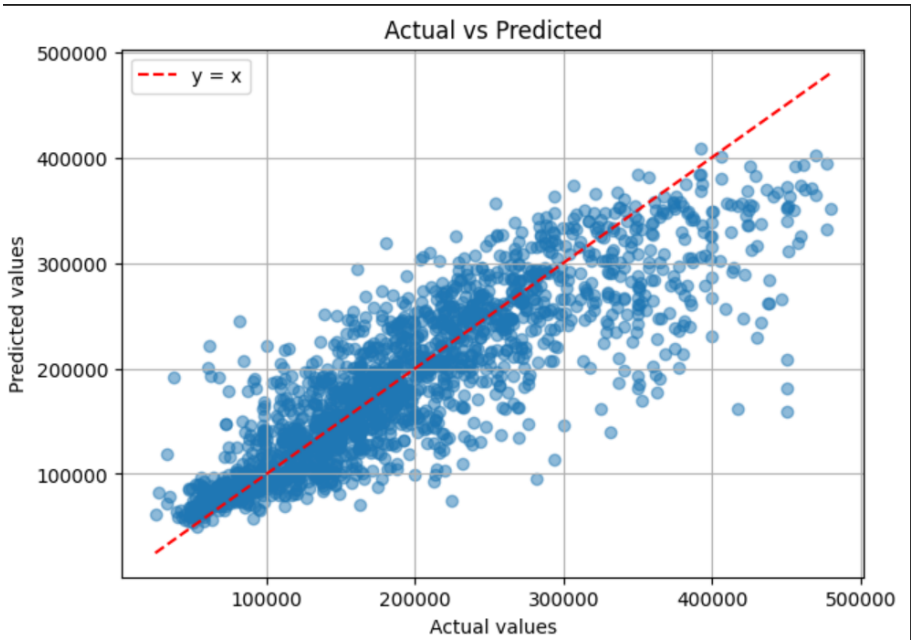

Figure 31: Actual vs predicted values for the improved neural network.

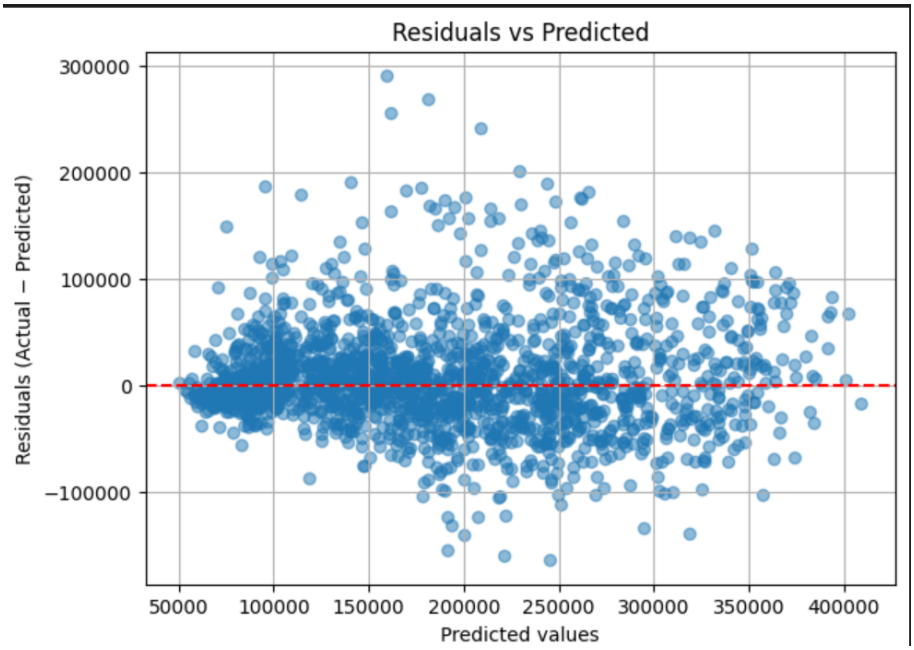

Figure 32: residuals for the improved neural network.

4 AutoEncoders

4–1 Getting data and performing appropriate preprocessing

We use the Fashion-MNIST dataset, which contains $60{,}000$ training and $10{,}000$ test grayscale images of clothing items, each of size 28×28. Images are loaded via `tensorflow.keras.d` and converted to floating point in the unit interval:

$$x \to \frac{x}{255.0}.$$

Standard preprocessing steps taken:

- Pixel values scaled to $[0, 1]$.
- Shapes converted to $(N, 28, 28, 1)$.

- Random shuffling and mini-batch creation performed using TensorFlow datasets during training.

Pixel normalization ensures that all input features (pixels) are within the same numerical range, preventing large gradients and improving the stability and convergence of training. Without normalization, higher raw pixel values could dominate the learning process, making optimization slower and less effective.

4–2 Implementing the AutoEncoder

The autoencoder is implemented as two sequential models: an *encoder* and a *decoder*. Key implementation choices:

- Input dimensionality: $28 \times 28 = 784$ flattened pixels.
- Encoder topology: a sequence of fully-connected layers whose widths are obtained by linearly interpolating between the input dimension and the chosen latent dimension. The encoder begins with a `Flatten()` layer followed by several `Dense` layers with ReLU activation.
- Latent dimensionality: 128.
- Decoder topology: mirrors the encoder by linearly interpolating from the latent space back to the input dimension. The final decoder layer outputs 784 units with ReLU activation and is reshaped to $(28, 28, 1)$.
- Activation: ReLU in encoder/decoder intermediate layers.
- Loss: Mean Squared Error (MSE)
- Optimizer: AdamW

4–3 Training the Network

The AutoEncoder was trained on the normalized Fashion-MNIST dataset using the following hyperparameters:

- **Batch size:** 64
- **Learning rate:** 0.001
- **Optimizer:** AdamW
- **Number of epochs:** 5

During training, the network processes the dataset in mini-batches, reconstructing each input image and computing the Mean Squared Error (MSE) between the original and reconstructed outputs. The gradients of this loss are propagated through both encoder and decoder networks, and parameters are updated using the AdamW optimizer.

The training progress can be visualized in Figure 33, which plots the reconstruction MSE per epoch.

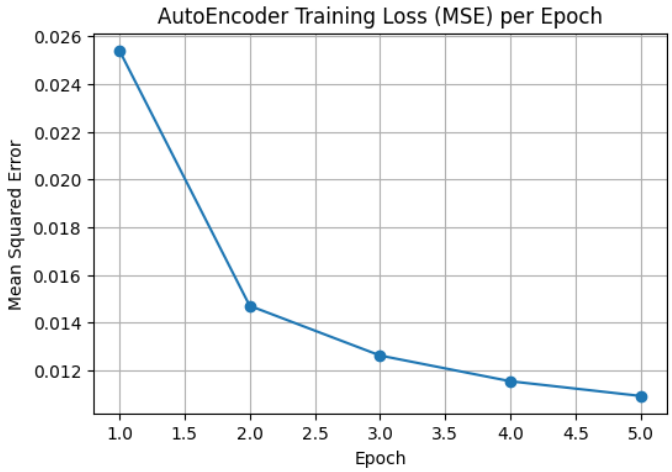

Figure 33: Mean Squared Error (MSE) per epoch during AutoEncoder training.

To qualitatively evaluate the AutoEncoder's performance, reconstructed images for five representative classes are shown in Figures 34–38. Each figure compares original and reconstructed examples.

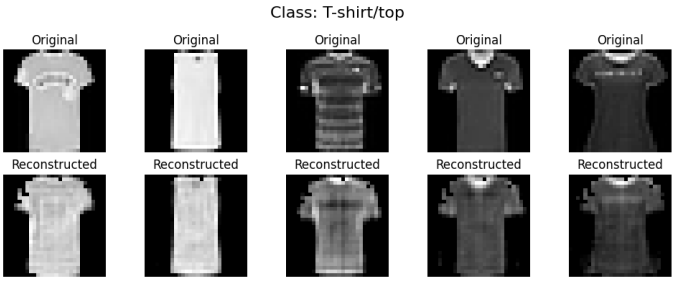

Figure 34: Original and reconstructed examples for the "T-shirt/top" class.

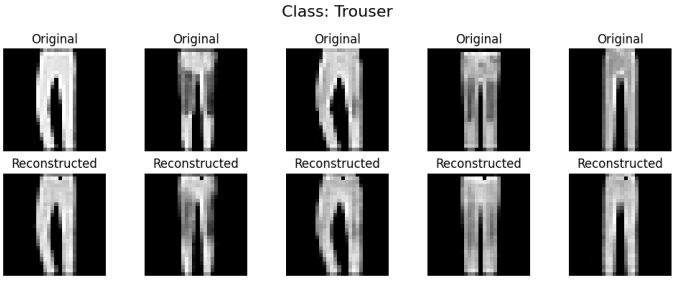

Figure 35: Original and reconstructed examples for the "Trouser" class.

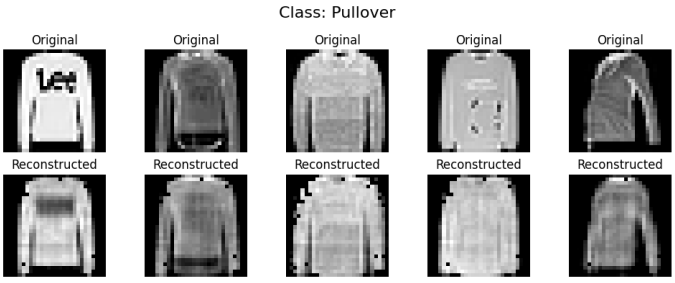

Figure 36: Original and reconstructed examples for the "Pullover" class.

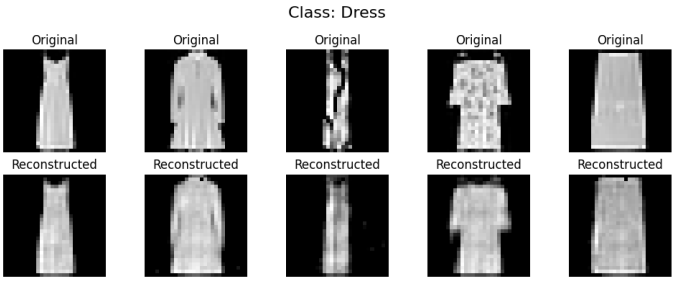

Figure 37: Original and reconstructed examples for the “Dress” class.

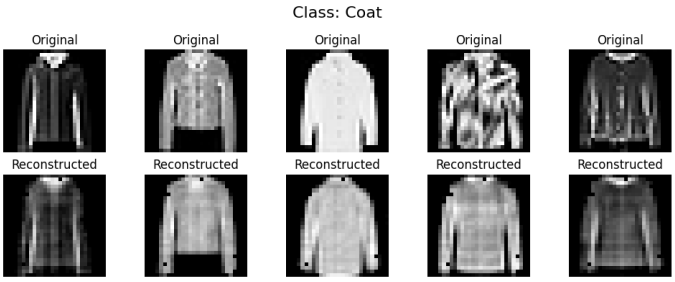

Figure 38: Original and reconstructed examples for the “Coat” class.

4–4 Evaluating the network’s performance

Evaluation was performed at two levels: (i) reconstruction quality and (ii) usefulness of the learned latent representation for tasks such as clustering and visualization.

4–4–1 Encoding the data

After training, test images were passed through the encoder to obtain latent vectors:

$$z_i = \text{Encoder}(x_i) \in \mathbb{R}^{128}.$$

These encoded representations are used for clustering and dimensionality reduction.

4–4–2 Clustering with K-Means

K-means clustering was applied to the latent vectors with $K = 10$ to match the ten classes in Fashion-MNIST.

4–4–3 Analyzing the clusters

The Confusion matrix for True Class vs. prediction by K-Means clusters is shown below:

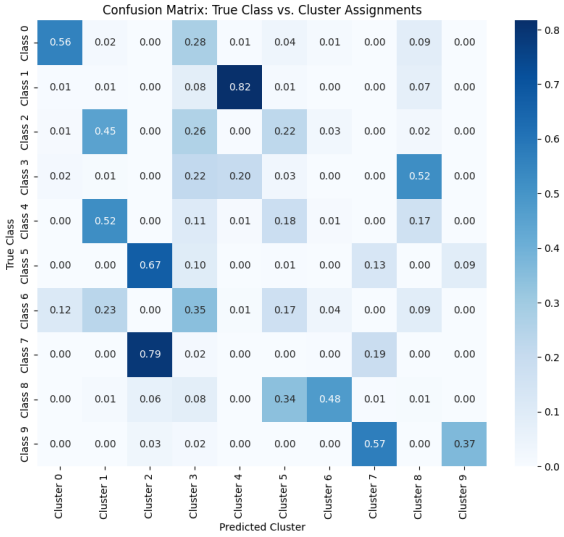

Figure 39: Confusion matrix for True Class vs. Cluster Assignments

To interpret the K-Means clusters, each cluster was assigned the most frequent true label among the samples belonging to it. This provides a mapping from the unsupervised cluster indices to actual Fashion-MNIST classes. The resulting mapping is shown below:

```
Cluster to Class Mapping:
{0: 0, 1: 4, 2: 7, 3: 6, 4: 1, 5: 8, 6: 8, 7: 9, 8: 3, 9: 9}
```

This indicates, for example, that cluster 0 predominantly represents class 0 (T-shirt/top), cluster 4 corresponds to class 1 (Trouser), and so on.

4–4–4 Dimensionality reduction and t-SNE

To visualize structure in the latent space, t-SNE was applied to the encoded vectors (2D projection). Two colored plots were produced:

- t-SNE colored by true Class labels.

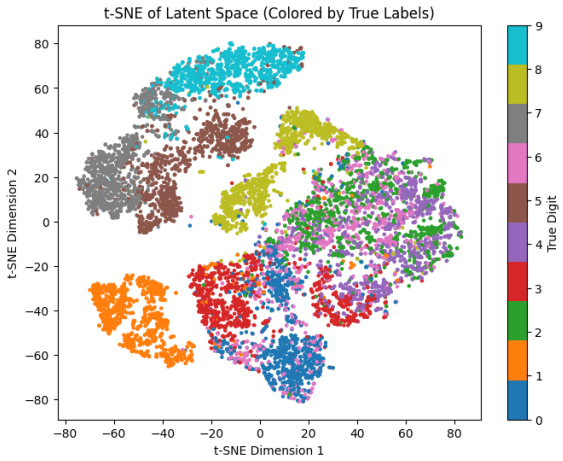

Figure 40: t-SNE of Latent Space

- t-SNE colored by cluster IDs.

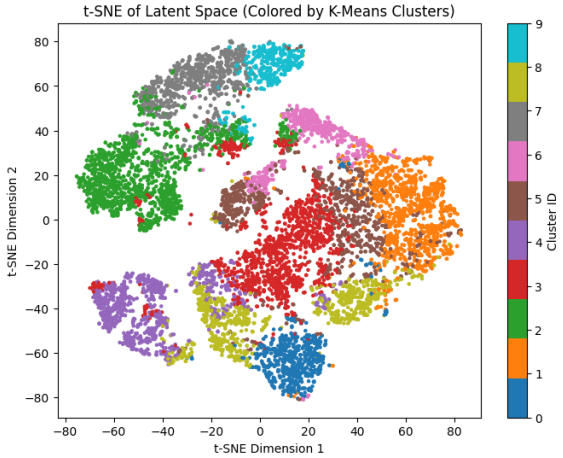

Figure 41: t-SNE of Latent Space (Colored by K-Means Clusters)

4–4–5 Comparison with raw data

For baseline comparison, K-means and t-SNE were also applied directly to flattened raw test images. This comparison highlights how the learned encoder compresses relevant information: clustering on encoded vectors typically yields tighter, more semantically-coherent clusters than clustering on raw pixels, because the encoder discards pixel-level noise and emphasizes class-discriminative structure.

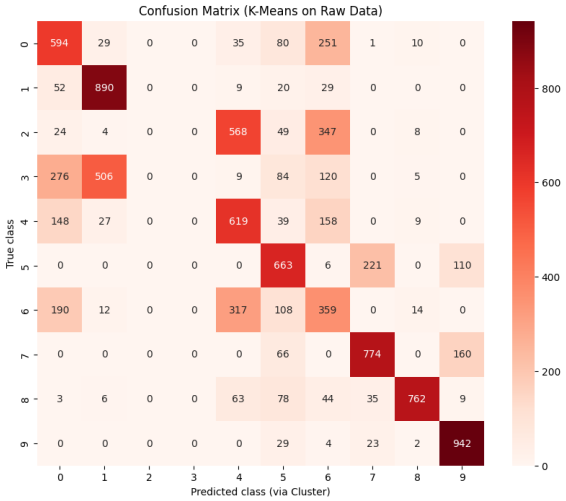

Figure 42: Confusion Matrix (K-Means on raw data)

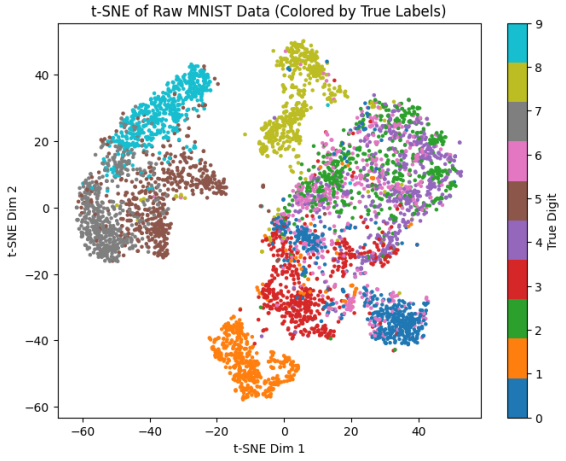

Figure 43: t-SNE of Raw MNIST Data (Colored by True Labels)

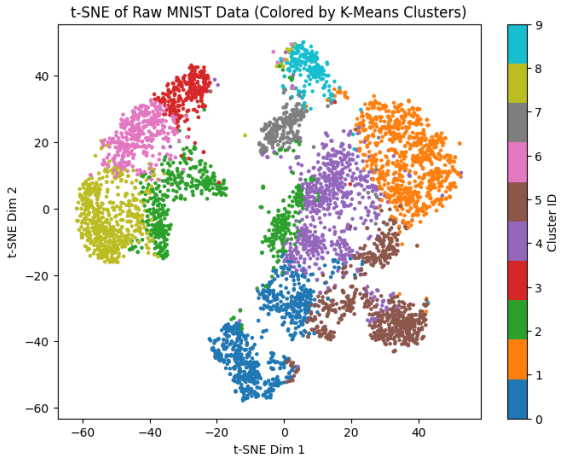

Figure 44: t-SNE of Raw MNIST Data (Colored by K-Means Clusters)

4–4–6 Evaluating clustering performance

Two external clustering metrics were computed to quantify similarity between cluster assignments obtained on encoded data and clusters from raw data:

- **Adjusted Rand Index (ARI)**: measures agreement between two partitions, corrected for chance.
- **Adjusted Mutual Information (AMI)**: mutual information between partitions normalized and adjusted for chance.
- ARI between clusters on encoded data and clusters on raw data: 0.54.
- AMI between clusters on encoded data and clusters on raw data: 0.68.