

Mohammad Jalili

40004073

HW2

# Question 1

## Part 1

When we use a ReLU activation in the second-to-last layer and a Sigmoid in the final layer for a binary classification task, the input to the Sigmoid will always be non-negative (because ReLU outputs values greater or equal to 0). As a result, the Sigmoid's output will always fall in the range  $[0.5, 1)$ , never dropping below 0.5. This behavior poses a problem for binary classification:

1. **No outputs below 0.5:** Because the Sigmoid never receives negative values, the network cannot produce probabilities less than 0.5. It effectively biases the model toward the positive class.
2. **Reduced representational flexibility:** Typically, for binary classification, you want the network to be free to produce logits (the inputs to the Sigmoid) that can span  $(-\infty, +\infty)$ . Negative logit values push the Sigmoid output toward 0, while positive values push it toward 1. A ReLU restricts these logits to  $[0, +\infty)$ , limiting the model's ability to learn an appropriate decision boundary.

Thus, having a ReLU in the second-to-last layer and a Sigmoid in the last layer for binary classification is typically *not* advisable because it constrains the output range and can hinder performance.

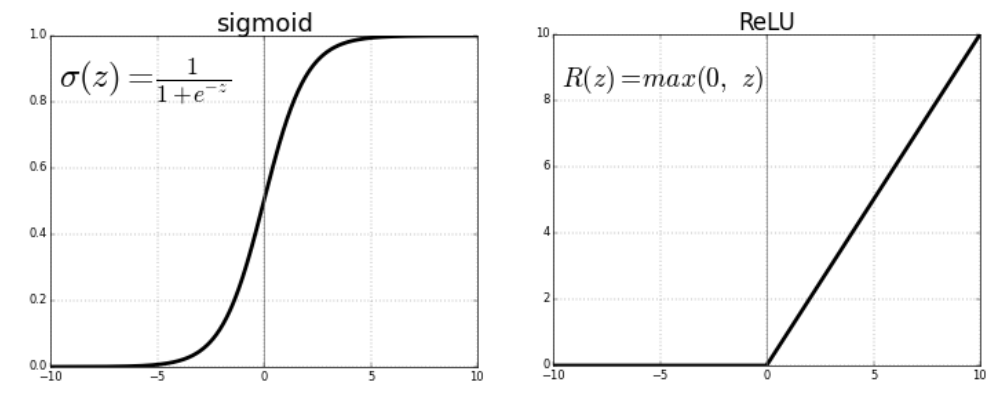


Figure 1: Plots of the Sigmoid and ReLU activation function.

## Part 2

$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0, \\ \alpha(e^x - 1), & \text{if } x < 0, \end{cases} \quad (1)$$

where  $\alpha$  is a hyperparameter.

## Gradient Calculation

$$\frac{d}{dx}\text{ELU}(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ \alpha e^x, & \text{if } x < 0. \end{cases}$$

## Advantage Over ReLU

One major advantage of ELU compared to ReLU is that it can produce negative outputs for negative inputs, rather than truncating them at zero. This helps alleviate the “dying ReLU” problem, in which a ReLU neuron can become permanently inactive for negative inputs. By allowing small negative outputs (and corresponding non-zero gradients), ELU can improve gradient flow during backpropagation and often leads to faster convergence in practice.

## Part 3

I have one perceptron for each side of the triangle, and then I combine them via a final perceptron that performs a logical AND.

A perceptron is a simple linear classifier defined by:

$$z = w_1x + w_2y + b$$

where:

$w_1$  and  $w_2$  are the weights for the inputs  $x$  and  $y$ ,

$b$  is the bias term,

$z$  is the linear output of the perceptron.

The activation function of the perceptron maps the linear output  $z$  to a binary value (0 or 1) as follows:

$$\text{Output} = \begin{cases} 1, & \text{if } z \geq 0, \\ 0, & \text{if } z < 0. \end{cases}$$

For the given triangle with vertices:

$$M_1(1, 0), \quad M_2(2, 2), \quad M_3(3, 0),$$

the edges of the triangle are represented by the following linear equations:

$$\begin{array}{ll} f_1(x, y) = y - 2x + 2 & \text{(Edge between } M_1 \text{ and } M_2), \\ f_2(x, y) = y + 2x - 6 & \text{(Edge between } M_2 \text{ and } M_3), \\ f_3(x, y) = y & \text{(Edge between } M_3 \text{ and } M_1). \end{array}$$

Each of these conditions can be modeled using a perceptron:

$$\begin{aligned}z_1 &= w_{11}x + w_{12}y + b_1 = -2x + y + 2, \\z_2 &= w_{21}x + w_{22}y + b_2 = 2x + y - 6, \\z_3 &= w_{31}x + w_{32}y + b_3 = y.\end{aligned}$$

The weights and biases for each perceptron are:

For  $f_1$ :  $w_{11} = -2$ ,  $w_{12} = 1$ ,  $b_1 = 2$ ,

For  $f_2$ :  $w_{21} = 2$ ,  $w_{22} = 1$ ,  $b_2 = -6$ ,

For  $f_3$ :  $w_{31} = 0$ ,  $w_{32} = 1$ ,  $b_3 = 0$ .

A point  $(x, y)$  satisfies all three conditions if:

$$\sigma(z_1) = 1, \quad \sigma(z_2) = 1, \quad \sigma(z_3) = 1,$$

where  $\sigma(z)$  is the step activation function.

The final output of the model, combining the three perceptrons, is obtained using an AND operation:

$$\text{Output} = \sigma(z_1) \cdot \sigma(z_2) \cdot \sigma(z_3).$$

The final mathematical form for determining if a point lies inside the triangle is:

$$\text{Output} = \sigma(-2x + y + 2) \cdot \sigma(2x + y - 6) \cdot \sigma(y).$$

## Perceptron Implementation

This code demonstrates my implementation of a perceptron with three activation functions: step, sigmoid, and ReLU, to classify points as inside or outside a triangle. I defined the triangle using three vertices and generated 2000 random points within a specified range. Using the `is_inside_triangle` function, I calculated whether each point lies inside the triangle by comparing the area of the triangle with the sum of the areas of sub-triangles formed with the point. The perceptron, which I designed to include customizable activation functions, evaluates each point based on these boundaries. For each activation function, I plotted the triangle and the points, with green indicating points inside and red for those outside.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Define the Perceptron class with multiple activation
  functions
5 class Perceptron:
6     def __init__(self, input_size, activation=None):
7         self.weights = np.random.rand(input_size + 1) # +1 for
          bias
```

```

8         self.activation = activation
9
10    def activation_function(self, x):
11        if self.activation == "step":
12            return 1 if x >= 0 else 0
13        elif self.activation == "sigmoid":
14            return 1 / (1 + np.exp(-x))
15        elif self.activation == "relu":
16            return max(0, x)
17        else:
18            raise ValueError("I have set 3 types of activation functions, but you still want something different. Are you trying to annoy me?")
19
20    def predict(self, inputs):
21        summation = np.dot(inputs, self.weights[1:]) + self.weights[0] # w.x + b
22        return self.activation_function(summation)
23
24
25
26
27 # Example usage
28 np.random.seed(73)
29
30 # Triangle vertices
31 triangle_vertices = np.array([[1, 0], [2, 2], [3, 0], [1, 0]])
32
33 # Generate 200 random points and labels
34 points = np.random.uniform(low=[0, -1], high=[4, 4], size=(2000, 2))
35
36
37 def is_inside_triangle(x, y):
38     p1, p2, p3 = np.array([1, 0]), np.array([2, 2]), np.array([3, 0])
39     def triangle_area(a, b, c):
40         return 0.5 * abs(a[0] * (b[1] - c[1]) + b[0] * (c[1] - a[1]) + c[0] * (a[1] - b[1]))
41     A = triangle_area(p1, p2, p3)
42     A1 = triangle_area(np.array([x, y]), p2, p3)
43     A2 = triangle_area(p1, np.array([x, y]), p3)
44     A3 = triangle_area(p1, p2, np.array([x, y]))
45     return 1 if np.isclose(A, A1 + A2 + A3) else 0
46
47
48 labels = np.array([is_inside_triangle(x, y) for x, y in points

```

```

49 ]
50 # Train Perceptron with different activation functions and
    visualize results
51 for activation in ["step", "sigmoid", "relu"]:
52     perceptron = Perceptron(input_size=2, activation=activation
        )
53     # Plot the results
54     plt.figure(figsize=(8, 8))
55     for point, label in zip(points, labels):
56         color = 'green' if label == 1 else 'red'
57         plt.scatter(point[0], point[1], color=color, alpha=0.6)
58     plt.plot(triangle_vertices[:, 0], triangle_vertices[:, 1],
        color='black', linestyle='--', linewidth=1.5)
59     plt.xlim(0, 4)
60     plt.ylim(-1, 4)
61     plt.title(f"Perceptron with {activation.capitalize()}
        Activation Function")
62     plt.xlabel("X-axis")
63     plt.ylabel("Y-axis")
64     plt.grid(True)
65     plt.show()

```

The figure below shows the output of the code.

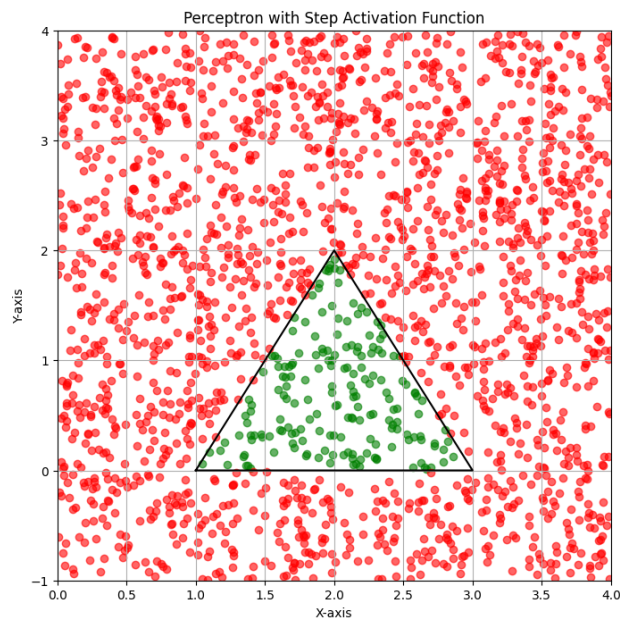


Figure 2: Output of perceptron neurons with Step Activation Function

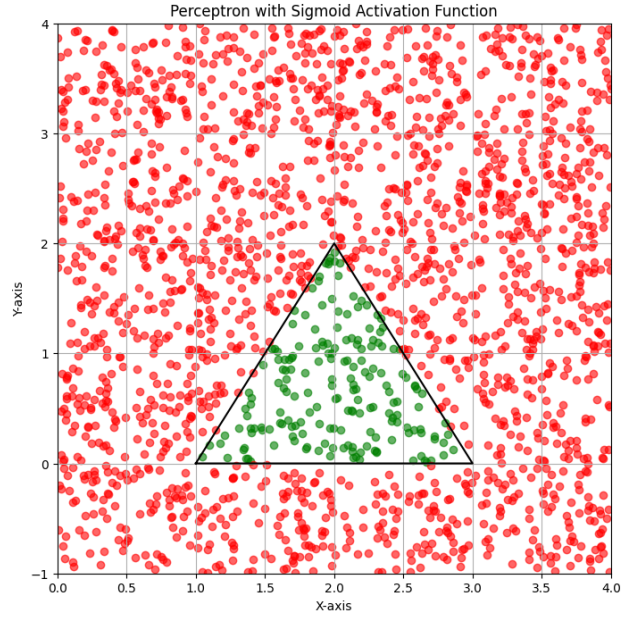


Figure 3: Output of perceptron neurons with Sigmoid Activation Function

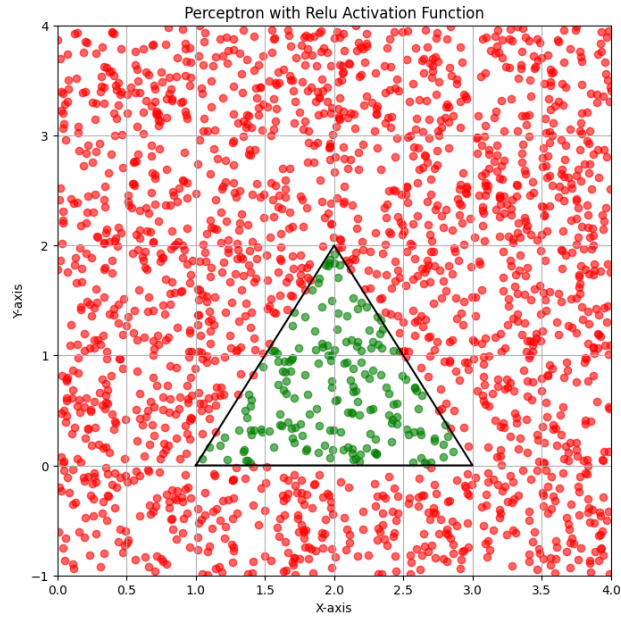


Figure 4: Output of perceptron neurons with ReLU Activation Function

## Explanation of Activation Functions

**Step Function** The step function is straightforward and binary, outputting either 0 or 1 depending on whether the input surpasses a threshold (typically 0). In the triangle problem, it creates clear and rigid decision boundaries, perfectly suitable for separating points inside and outside the triangle. While efficient and simple to implement, it struggles with noisy data or points near the edges of the triangle, as its rigid nature

leaves no room for uncertainty.

**Sigmoid Function** The sigmoid function maps inputs to a continuous range between 0 and 1, making it ideal for probabilistic interpretations. In the triangle problem, it provides smooth decision boundaries, which are helpful for handling points near the edges of the triangle or in cases where the data is noisy. However, its computational complexity is higher than the step function, and it may face challenges with very large or small inputs due to gradient saturation, potentially slowing learning.

**ReLU Function** ReLU outputs the input directly if it is positive, and zero otherwise, making it effective for learning complex, non-linear relationships. For the triangle problem, it can create flexible and interpretable decision boundaries, particularly when expanding the problem to more complex scenarios. However, it may misclassify points near the edges due to its sensitivity to negative values, and "dead neurons" (where weights stop updating for negative outputs) can occur.

### Comparison Table

Activation Function	Key Feature	Advantages	Disadvantages	Suitability for Triangle Problem
<b>Step Function</b>	Binary output (0 or 1)	Simple, fast, and creates sharp decision boundaries.	Rigid; not suitable for noisy data or points near boundaries.	Ideal for binary classification (inside or outside).
<b>Sigmoid Function</b>	Smooth output (0 to 1)	Provides probabilistic output; handles uncertainty near boundaries well.	Computationally expensive; may face gradient saturation issues.	Good for noisy or uncertain data.
<b>ReLU Function</b>	Linear for positives	Flexible and effective for complex, non-linear relationships; fast convergence.	Sensitive to negative values; prone to "dead neurons" for negative inputs.	Suitable for extended or more complex scenarios.

Table 1: Comparison of Activation Functions in the Triangle Problem



## Question 2

### teleCust1000t Dataset

The *teleCust1000t* dataset contains information about 1000 customers of a telecommunication company. Each row corresponds to a single customer and includes demographic details, subscription history, and usage patterns. The target variable, *custcat*, classifies customers into four categories, such as *Basic Service*, *E-Service*, and so on. Understanding these categories is crucial for tailoring marketing strategies and improving customer retention.

### Initial Exploratory Analysis

I began by loading the dataset and examining its structure:

```
[5] df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 12 columns):
#   Column      Non-Null Count  Dtype  
---  -
0   region      1000 non-null   int64  
1   tenure      1000 non-null   int64  
2   age         1000 non-null   int64  
3   marital     1000 non-null   int64  
4   address     1000 non-null   int64  
5   income      1000 non-null   float64 
6   ed          1000 non-null   int64  
7   employ      1000 non-null   int64  
8   retire      1000 non-null   float64 
9   gender      1000 non-null   int64  
10  reside      1000 non-null   int64  
11  custcat     1000 non-null   int64  
dtypes: float64(2), int64(10)
memory usage: 93.9 KB
```

Figure 5: teleCust1000t Dataset

The dataset has 1000 records (rows) and several columns (features). Most columns are numeric (*e.g.*, *age*, *tenure*, *income*), while some are categorical or binary (such as *marital*, *custcat*).

```
print(df.isnull().sum())  
  
region      0  
tenure      0  
age         0  
marital     0  
address     0  
income     0  
ed         0  
employ     0  
retire     0  
gender     0  
reside     0  
custcat    0  
dtype: int64
```

Figure 6: teleCust1000t Dataset

A quick inspection with `df.isnull().sum()` revealed no significant missing data.

## Data Distribution Analysis

First, I plotted the data distribution.

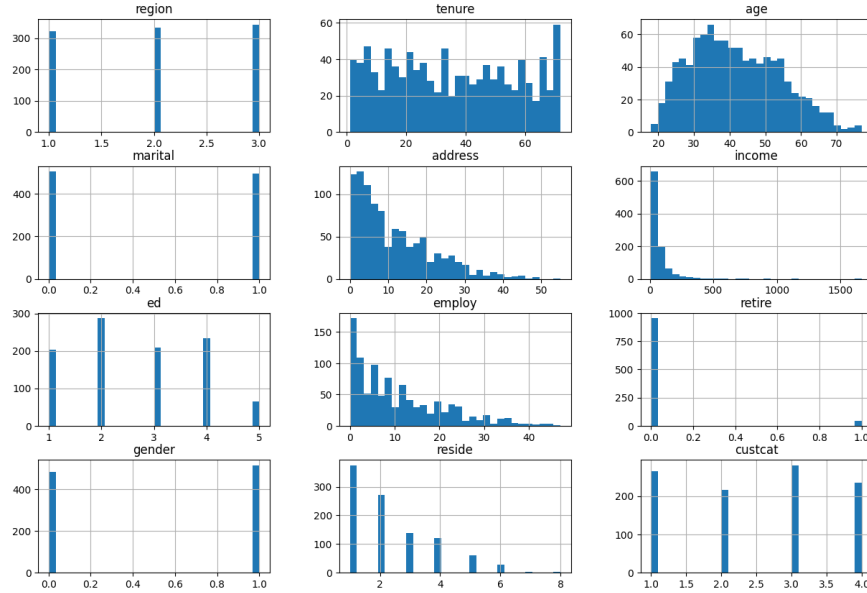


Figure 7: teleCust1000t Dataset Distribution

**region:** The values are discrete (1, 2, 3), and each category appears to have a relatively balanced count. This suggests that customers are somewhat evenly distributed across different regions.

**tenure:** The range spans from 0 up to around 70 months. Interestingly, the distribution seems to have two peaks: one for shorter tenures (under 10 months) and another around 60+ months, indicating both relatively new subscribers and long-term loyal customers.

**age:** Values range from about 18 to 79, with a prominent peak between 30 and 40. The curve slightly skews to the right, suggesting fewer older customers as age increases.

**marital:** The data seems binary (0 or 1). A larger count of one category indicates most customers share a similar marital status, but exact interpretation depends on which label stands for married vs. single.

**address:** Spanning roughly 0 to 50, this feature is heavily skewed right: many customers have low values (perhaps less time at a current address), while a few have very high values (over 40), suggesting long-term residency.

**income:** Ranging from near 0 to well above 1500, but the majority of values lie under a few hundred. This is also strongly right-skewed, indicating that while most customers have modest incomes, there is a small subset with significantly higher incomes.

**ed:** This takes discrete values (1, 2, 3, 4, 5), likely representing education levels. Middle categories seem most common, whereas higher levels (like 5) appear less frequent.

**employ:** Ranging up to around 40, it might represent years of employment. The distribution shows most people are in lower or medium ranges, with only a few having very long employment histories.

**retire:** This appears to be binary (0 or 1). The histogram indicates the vast majority of individuals are *not* retired, with only a small portion being retirees.

**gender:** Another binary feature. Both categories are represented, but it looks like there is a slight imbalance favoring one category over the other.

**reside:** Possibly signifying the number of different residences or years at a location, it ranges from 0 to around 8. Again, a right-skew is evident, as most customers are concentrated at the lower end.

**custcat:** The target variable with four categories (1, 2, 3, 4). The histogram shows a fairly balanced distribution among these classes, with some slight variation in counts. This balance is typically beneficial for classification tasks.

## Correlation Matrix Analysis

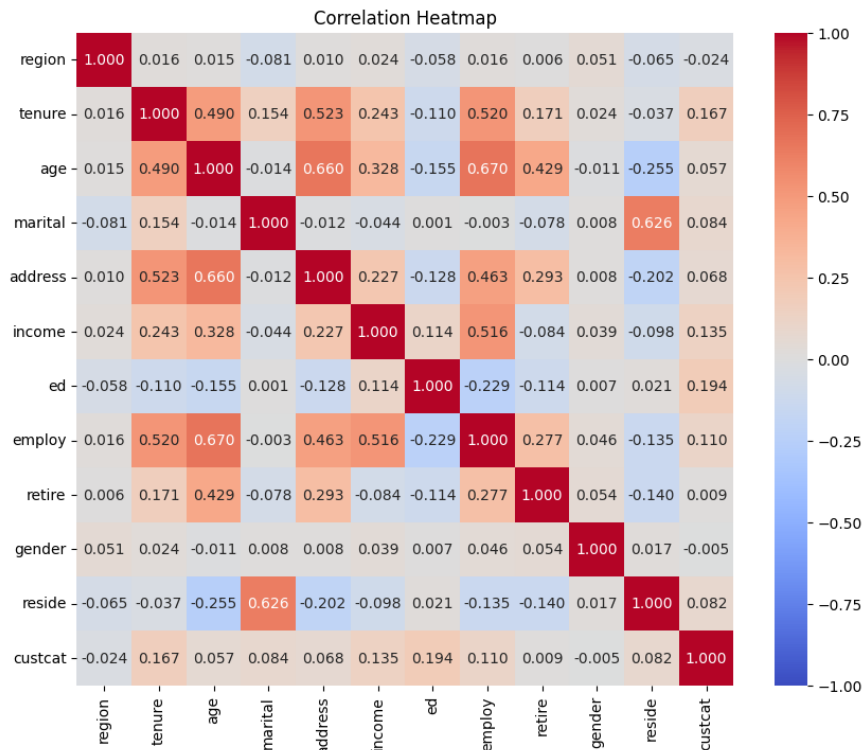


Figure 8: Correlation Matrix Analysis

### High Correlation Among Age, Employ, and Address:

You can see that *age* correlates strongly with both *employ* (around 0.67) and *address* (around 0.66). This makes intuitive sense since older customers often have more years of employment and may have lived longer at a single address.

### Tenure's Moderate Associations:

*Tenure* has a moderate positive correlation (about 0.49) with *age* and around 0.52 with *employ*. This suggests that older or more experienced customers tend to remain with the telecommunications service for a longer period.

### reside & marital:

The correlation between *reside* and *marital* is around 0.63, indicating that married customers may be more likely to have stable or fewer relocations. However, interpretation depends on the exact definition of these variables.

### Weak to Moderate Correlations with custcat:

Our target variable, *custcat*, shows the highest correlations (though still not very large) with features like *tenure* ( $\sim 0.17$ ), *ed* ( $\sim 0.19$ ), and *income* ( $\sim 0.14$ ). These values suggest that while these features offer some predictive signal, they are far from being definitive indicators of the customer category.

### Overall Skew Toward Lower Correlations:

Most pairwise correlations range between -0.2 and +0.7. Only a few pairs exceed 0.6 (for instance, *age-employ* and *age-address*). The lack of extremely high correlations (e.g., above 0.8) is good news for avoiding multicollinearity issues in linear models. At the same time, it means we likely need to combine multiple features (potentially in a nonlinear way) to achieve strong predictive power.

## Histogram Analysis of Income and Tenure

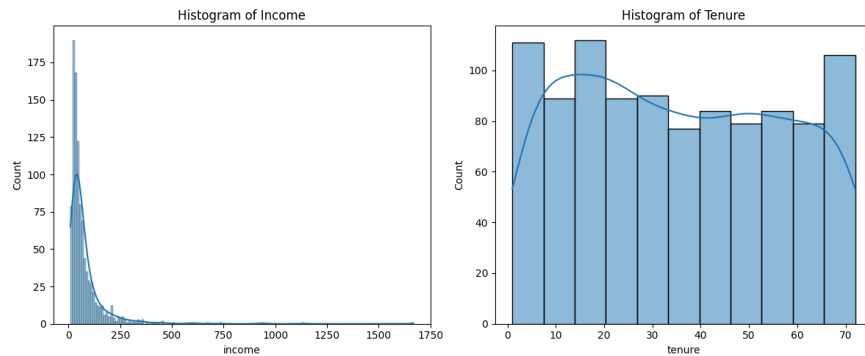


Figure 9: Histogram Analysis of Income and Tenure

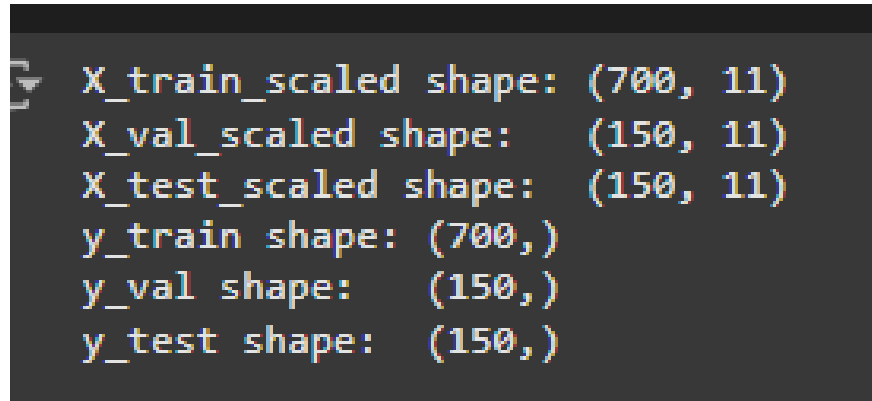
Figure 16 shows histograms of the `income` and `Tenure` feature. For `income`, the distribution is heavily right-skewed, with most customers clustered in the range of 0–50. As the income grows larger (beyond 200), the frequency rapidly decreases, indicating only a small minority of high-income customers. This long tail highlights the presence of outliers or a subset of individuals whose annual income is substantially higher than the majority of the population.

For `Tenure`, the histogram does not resemble a classic bell-shaped distribution; instead, it appears to have several local peaks. Notably, there is a considerable fraction

of customers in their first 10 months, as well as a substantial group of long-term subscribers (above 60 months). The mid-range (20–40 months) is also populated, albeit at a slightly lower frequency than the initial peak.

## Data normalization with MinMaxScaler

I allocated 70% of the data for training, 15% for testing, and 15% for validation.



```
X_train_scaled shape: (700, 11)
X_val_scaled shape: (150, 11)
X_test_scaled shape: (150, 11)
y_train shape: (700,)
y_val shape: (150,)
y_test shape: (150,)
```

Figure 10: Histogram Analysis of Income and Tenure

## Models

### Model with one hidden layer

I designed a simple neural network with one hidden layer to classify data into four classes. The architecture includes:

**Input Layer:** Matches the feature size of the dataset.

**Hidden Layer:** A fully connected layer with variable neurons (16, 32, 64, 128 across models).

**Activation Function:** ReLU applied to the hidden layer's output.

**Output Layer:** A fully connected layer producing logits for classification.

The model was trained using the following configuration:

**Optimizer:** Stochastic Gradient Descent (SGD) with momentum.

**Loss Function:** Categorical Cross-Entropy.

**Performance Metrics:** Training and validation loss, along with accuracy.

I trained four models with different hidden layer sizes and compared their performance based on validation loss. The best-performing model, identified by the lowest validation loss, was further analyzed using a confusion matrix and a detailed classification report.

Hidden Size	Train Loss	Validation Loss	Train Accuracy	Validation Accuracy
16	1.2401	1.2698	0.4314	0.3867
32	1.2352	1.2674	0.4343	0.3933
64	1.2212	1.2616	0.4386	0.4000
128	<b>1.2157</b>	<b>1.2606</b>	<b>0.4414</b>	<b>0.4067</b>

Table 2: Performance metrics for models with varying hidden layer sizes.

As you can see in the table, as the number of neurons increases, the loss decreases and the accuracy of the model also increases.

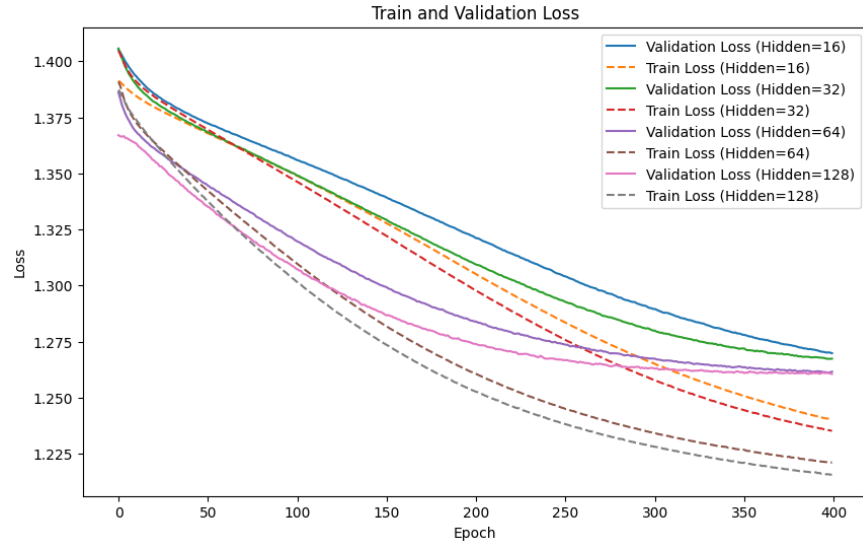


Figure 11: Train and Validation Loss for Model with one hidden layer

The plot illustrates the performance of models with varying hidden layer sizes during training. From the above plot we can extract the following points and

- (a) **Learning and Convergence:** All models demonstrate a steady decline in training and validation loss as training progresses, indicating effective learning. Toward the final epochs, the losses plateau, suggesting that the models have converged and are no longer improving significantly.
- (b) **Impact of Hidden Layer Size:** Models with larger hidden layers (e.g., 128 neurons) consistently achieve lower training and validation losses, indicating better capacity to learn complex patterns. Smaller hidden layers (e.g., 16 neurons) result in higher losses, likely due to limited capacity to model the dataset effectively.
- (c) **Balance Between Training and Validation Loss:** The gap between training and validation losses is relatively small across all models, showing good generalization and minimal overfitting. Models with smaller hidden layers exhibit slightly

higher validation losses, suggesting potential underfitting due to insufficient model complexity.

- (d) **Best Model:** The model with 128 neurons in the hidden layer achieves the lowest validation loss and remains stable across epochs, making it the best performer among the tested configurations.

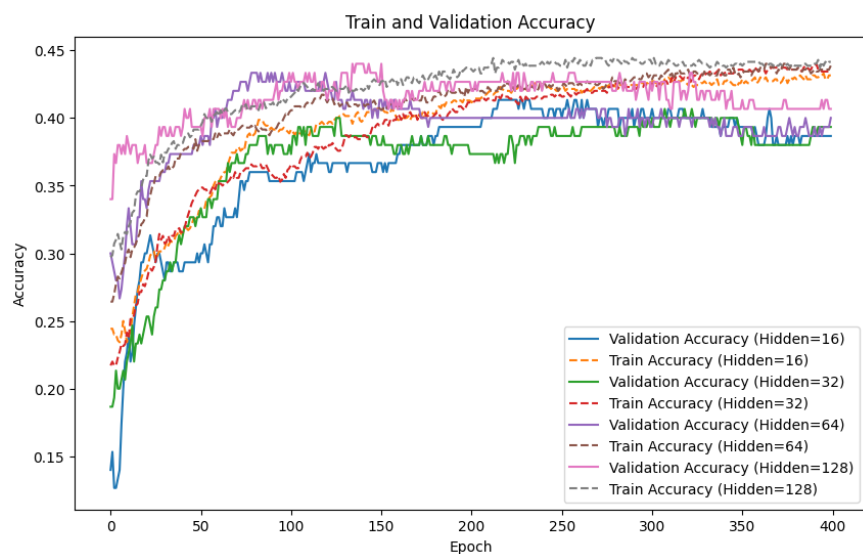


Figure 12: Train and Validation Loss for Model with one hidden layer



## Model with 2 hidden layers

As you can see from the graph, the model with 2 hidden layers has more learning capacity than one layer.

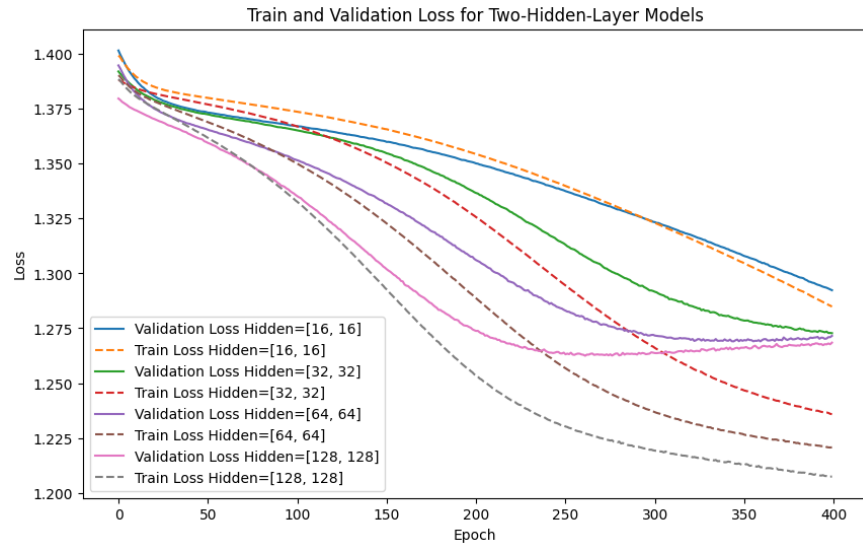


Figure 13: Train and Validation Loss for Model with one hidden layer

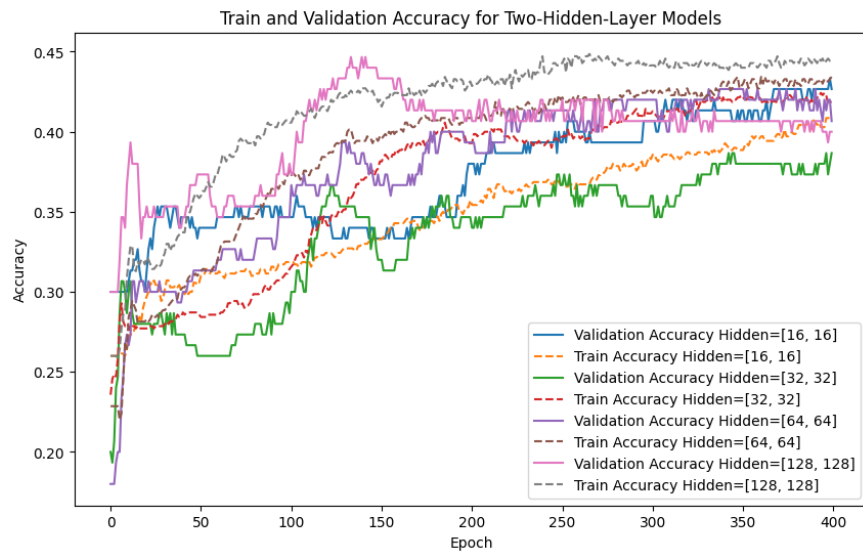


Figure 14: Train and Validation Accuracy for Model with 2 hidden layer

## Effect of adding batch normalization

Adding batch normalization to a network resulted in faster and more stable training, improved generalization ability, and could often lead to higher accuracy by reducing

the problem of "internal variable shifting" where the distribution of inputs to each layer changes significantly during training.

## Add a dropout layer

Adding a dropout layer increased the stability of the network and reduced fluctuations during the training process. It also reduced the likelihood of overfitting.

## ADOPT

Using the ADOPT optimizer reduced the loss in the 2-layer model, but no specific changes were added to the attack layer model.

Next, 10 test samples were selected and tested with both models, but the first model performed better than the second model, while we had seen in the graphs that these two models performed closely. One of the reasons for this discrepancy could be that model 2 did not know enough about the existing pattern. Of course, this was a coincidence here. For example, if we repeat this task again, different results will be obtained.

```
5] from torch.utils.data import DataLoader, TensorDataset, SubsetRandomSampler

indices = torch.randperm(len(X_train_tensor))[:10]
sampler = SubsetRandomSampler(indices)
sample_loader = DataLoader(TensorDataset(X_train_tensor, y_train_tensor), batch_size=1, sampler=sampler)

# Function to evaluate a model on a data loader
def evaluate_model(model, data_loader):
    model.eval()
    y_true, y_pred = [], []
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            outputs = model(X_batch)
            _, predicted = torch.max(outputs, 1)
            y_true.extend(y_batch.tolist())
            y_pred.extend(predicted.tolist())
    return y_true, y_pred

# Test model_1 (assuming model_1["model"] is the trained model)
print("Model 1 Evaluation:")
y_true_1, y_pred_1 = evaluate_model(model_1["model"], sample_loader)
print(f"True Labels: {y_true_1}")
print(f"Predicted Labels: {y_pred_1}")

# Test model_2 (assuming model_2["model"] is the trained model)
print("\nModel 2 Evaluation:")
y_true_2, y_pred_2 = evaluate_model(model_2["model"], sample_loader)
print(f"True Labels: {y_true_2}")
print(f"Predicted Labels: {y_pred_2}")

Model 1 Evaluation:
True Labels: [1, 0, 1, 3, 2, 1, 2, 1, 3, 1]
Predicted Labels: [1, 0, 1, 3, 2, 0, 0, 0, 1, 1]

Model 2 Evaluation:
True Labels: [1, 2, 2, 1, 0, 3, 1, 3, 1, 1]
Predicted Labels: [3, 0, 2, 0, 0, 1, 1, 3, 3, 0]
```

Figure 15: Output of 2 final models for 10 random data from the test set

## Ensembling Method

In this method, I combine the predictions of two trained models (`model_1` and `model_2`) by averaging their outputs for the test data. For each batch in the test loader, I let both models generate predictions and then compute the element-wise average of their outputs:

$$\text{Ensemble\_Outputs} = \frac{\text{Outputs\_1} + \text{Outputs\_2}}{2}$$

Next, I determine the final class prediction by selecting the class with the highest averaged probability:

$$\text{Prediction} = \arg \max(\text{Ensemble\_Outputs})$$

This ensembling technique, called *output averaging*, allows me to reduce errors from individual models and achieve better overall accuracy. I find this approach particularly effective when the models are diverse, as it combines their strengths and offsets their weaknesses. Finally, I evaluate the ensemble's performance using accuracy, a confusion matrix, and a classification report.

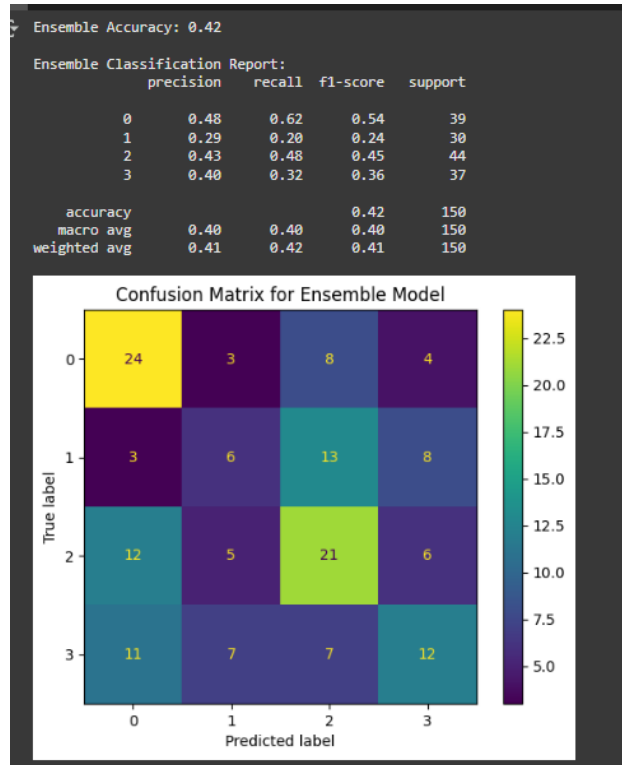


Figure 16: Ensemble model evaluation results

## Quseion 3

### main code

This function, named `convertImageToBinary`, takes the input image and converts it into a binary vector based on pixel intensities. In this binary vector, white pixels (with high intensity) are represented by -1, and black pixels (with low intensity) are represented by 1.

```
1 from PIL import Image, ImageDraw
2 import random
3
4 def convertImageToBinary(path):
5     """
6     Convert an image to a binary representation based on pixel
7     intensity.
8
9     Args:
10         path (str): The file path to the input image.
11
12     Returns:
13         list: A binary representation of the image where white
14             is represented by -1 and black is represented by 1.
15     """
16     # Open the image file.
17     image = Image.open(path)
18
19     # Create a drawing tool for manipulating the image.
20     draw = ImageDraw.Draw(image)
21
22     # Determine the image's width and height in pixels.
23     width = image.size[0]
24     height = image.size[1]
25
26     # Load pixel values for the image.
27     pix = image.load()
28
29     # Define a factor for intensity thresholding.
30     factor = 100
31
32     # Initialize an empty list to store the binary
33     # representation.
34     binary_representation = []
35
36     # Loop through all pixels in the image.
37     for i in range(width):
38         for j in range(height):
```

```

36         # Extract the Red, Green, and Blue (RGB) values of
           the pixel.
37         red = pix[i, j][0]
38         green = pix[i, j][1]
39         blue = pix[i, j][2]
40
41         # Calculate the total intensity of the pixel.
42         total_intensity = red + green + blue
43
44         # Determine whether the pixel should be white or
           black based on the intensity.
45         if total_intensity > (((255 + factor) // 2) * 3):
46             red, green, blue = 255, 255, 255 # White pixel
47             binary_representation.append(-1)
48         else:
49             red, green, blue = 0, 0, 0 # Black pixel
50             binary_representation.append(1)
51
52         # Set the pixel color accordingly.
53         draw.point((i, j), (red, green, blue))
54
55     # Clean up the drawing tool.
56     del draw
57
58     # Return the binary representation of the image.
59     return binary_representation

```

The second cell features two main functions. The `generateNoisyImages()` function specifies a list of several images and, for each one, calls the `getNoisyBinaryImage()` function to add random noise to every single pixel, saving the noisy output under a new name. The `getNoisyBinaryImage()` function first opens the input image, and using a nested loop over the width and height, generates a random noise value for each pixel and adds it to the pixel's RGB channels. Then, the final value of each color channel is clamped to the range `[0, 255]` to prevent oversaturation or negative values. Finally, the noisy image is saved to the specified output path.

```

1 from PIL import Image, ImageDraw
2 import random
3
4 def generateNoisyImages():
5     # List of image file paths
6     image_paths = [
7         "/content/1.jpg",
8         "/content/2.jpg",
9         "/content/3.jpg",
10        "/content/4.jpg",
11        "/content/5.jpg"
12    ]

```

```

13
14     for i, image_path in enumerate(image_paths, start=1):
15         noisy_image_path = f"/content/noisy{i}.jpg"
16         getNoisyBinaryImage(image_path, noisy_image_path)
17         print(f"Noisy image for {image_path} generated and saved as {noisy_image_path}")
18
19 def getNoisyBinaryImage(input_path, output_path):
20     """
21     Add noise to an image and save it as a new file.
22
23     Args:
24         input_path (str): The file path to the input image.
25         output_path (str): The file path to save the noisy
26                             image.
27     """
28     # Open the input image.
29     image = Image.open(input_path)
30
31     # Create a drawing tool for manipulating the image.
32     draw = ImageDraw.Draw(image)
33
34     # Determine the image's width and height in pixels.
35     width = image.size[0]
36     height = image.size[1]
37
38     # Load pixel values for the image.
39     pix = image.load()
40
41     # Define a factor for introducing noise.
42     noise_factor = 10000000
43
44     # Loop through all pixels in the image.
45     for i in range(width):
46         for j in range(height):
47             # Generate a random noise value within the
48             # specified factor.
49             rand = random.randint(-noise_factor, noise_factor)
50
51             # Add the noise to the Red, Green, and Blue (RGB)
52             # values of the pixel.
53             red = pix[i, j][0] + rand
54             green = pix[i, j][1] + rand
55             blue = pix[i, j][2] + rand
56
57             # Ensure that RGB values stay within the valid
58             # range (0-255).

```

```
55         if red < 0:
56             red = 0
57         if green < 0:
58             green = 0
59         if blue < 0:
60             blue = 0
61         if red > 255:
62             red = 255
63         if green > 255:
64             green = 255
65         if blue > 255:
66             blue = 255
67
68         # Set the pixel color accordingly.
69         draw.point((i, j), (red, green, blue))
70
71     # Save the noisy image as a file.
72     image.save(output_path, "JPEG")
73
74     # Clean up the drawing tool.
75     del draw
76
77 # Generate noisy images and save them
78 generateNoisyImages()
```

## Network Design

I designed a Hamming network tailored for pattern recognition and noise tolerance. Initially, I converted input images into binary patterns, where each pixel was represented as either +1 or -1 based on a threshold value. I trained the network by normalizing and storing these patterns as weight matrices. During the testing phase, I introduced varying levels of noise into the input patterns by randomly flipping a percentage of their binary values. The network then utilized the pre-trained weights to recall the closest matching pattern by computing similarity scores between the noisy input and stored patterns. As I increased the noise levels, I observed the network's robustness and its threshold of failure, where the accuracy of pattern recovery significantly degraded. This approach effectively demonstrated the network's capacity to handle noisy data.

as noise increases beyond 30, accuracy begins to drop, with the network still achieving a respectable 92.05 accuracy at 70 noise. This demonstrates the network's resilience to noise, though its effectiveness diminishes at higher noise levels.

Table 3: Average Accuracy of the Hamming Network at Different Noise Levels (Dataset 2)

Noise Level (%)	Average Accuracy (%)
10	100.00
20	100.00
30	100.00
40	96.50
50	95.39
70	92.05

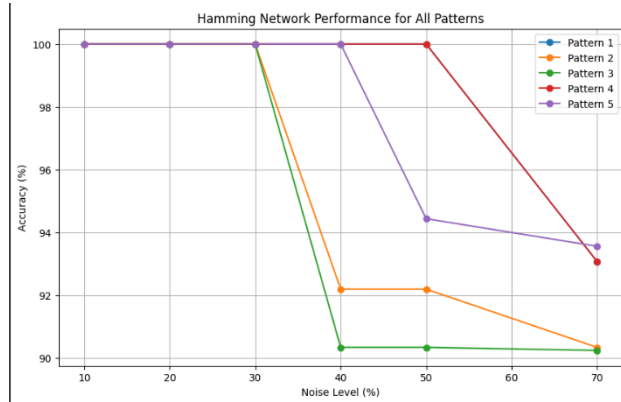


Figure 17: Hamming Network Performance for All Patterns



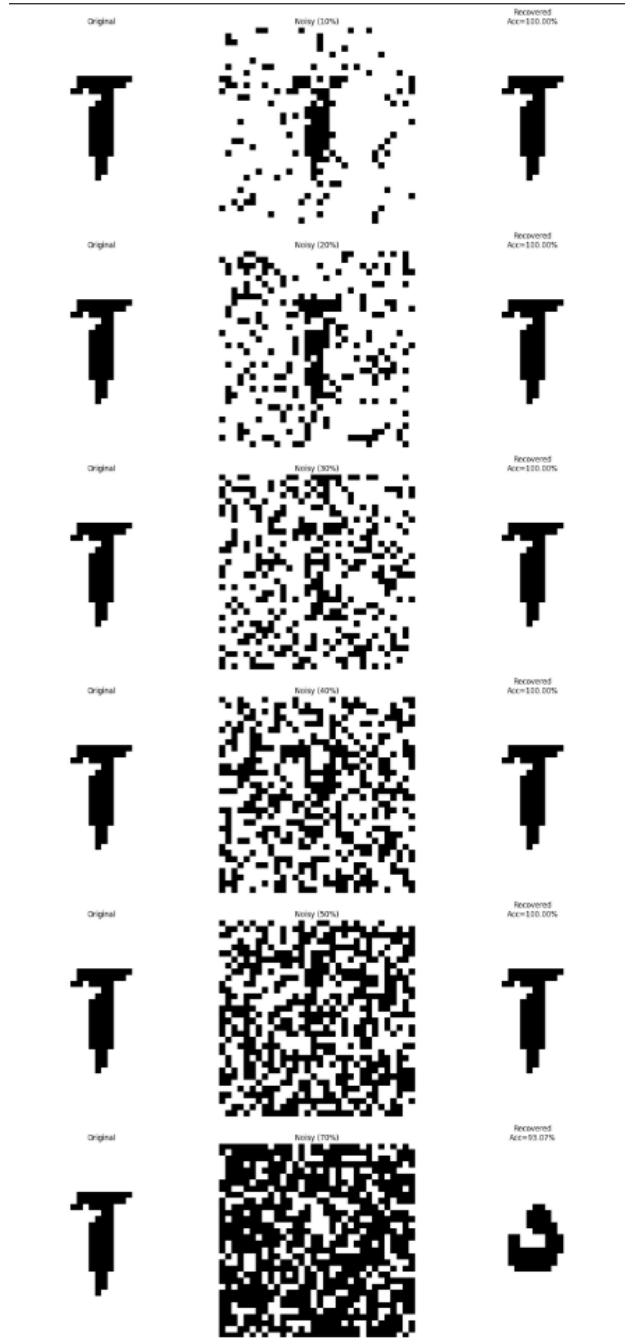


Figure 18: Hamming network performance for pattern 1

## Part 2

Now I'll examine the performance of the model by adding missing points.

## Hamming Network Analysis with Missing Points

### Pattern-Wise Analysis

Patterns 1, 4, and 5 maintain 100% accuracy across all levels of missing points, showcasing their robustness. However, patterns 2 and 3 exhibit slight declines in accuracy, particularly at higher missing levels. This indicates that some patterns are more resilient to missing points than others, potentially due to differences in their complexity or structure.

### Missing Points Percentage-Wise Analysis

The network shows robust performance, maintaining 100% accuracy up to 50% missing points. As the missing percentage increases beyond 50%, the accuracy starts to decline gradually. At 80% missing points, the average accuracy drops to approximately 95.79%. This trend highlights the network's ability to handle moderate levels of missing data effectively but shows its limitations under severe data loss.

Table 4: Average Accuracy for Each Pattern

Pattern	Average Accuracy (%)
1	100.00
2	98.96
3	91.79
4	100.00
5	100.00

Table 5: Average Accuracy for Each Missing Points Percentage

Missing Points (%)	Average Accuracy (%)
10	100.00
20	100.00
30	100.00
40	100.00
50	100.00
60	98.29
70	97.86
80	95.79



Figure 19: Enter Caption

## Why Does Accuracy Decrease with Missing Points?

As the number of missing points in the input pattern increases, the accuracy of the Hamming network decreases. This is because missing points reduce the amount of available information in the input pattern, making it harder for the network to correctly recall the original pattern. With fewer valid data points, the similarity computation between the input and stored patterns becomes less reliable. Consequently, the network struggles to match the incomplete input to its stored patterns, leading to lower accuracy as the missing ratio increases.

## Potential Solutions to Mitigate Accuracy Drop

To address the decline in accuracy as missing points increase, the following strategies can be employed:

**Noise-Tolerant Models:** Modify the Hamming network to assign higher weights to valid (non-missing) points during similarity computations. This ensures the network relies more on available information.

**Data Preprocessing:** Preprocess the input patterns to fill missing points with imputed values (e.g., nearest-neighbor or mean-value imputation) before feeding them into the network.

**Error Correction Codes:** Incorporate error correction codes into the network design to better handle noisy or incomplete inputs.

**Hybrid Approaches:** Combine the Hamming network with other robust techniques, such as probabilistic models or deep learning methods, to improve pattern recovery under severe data loss.

These approaches can enhance the network's robustness, allowing it to perform well even when significant portions of the input are missing.

## Question 4

The dataset must be standardized using `StandardScaler`, and the data should be split into training, validation, and test sets.

The following Python code demonstrates the preprocessing of the California Housing dataset and prepares it for training:

### Data Preprocessing

```
1 # Load and preprocess the California Housing dataset
2 from sklearn.datasets import fetch_california_housing
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import StandardScaler
5 import torch
6
7 # Load the dataset
8 data = fetch_california_housing()
9 X, y = data.data, data.target
10
11 # Standardize features
12 scaler = StandardScaler()
13 X = scaler.fit_transform(X)
14
15 # Split the dataset into train, validation, and test sets (70%,
16     15%, 15%)
17 X_train, X_temp, y_train, y_temp = train_test_split(X, y,
18     test_size=0.3, random_state=73)
19 X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
20     test_size=0.5, random_state=73)
21
22 # Convert data to PyTorch tensors
23 X_train = torch.tensor(X_train, dtype=torch.float32)
24 y_train = torch.tensor(y_train, dtype=torch.float32).view(-1,
25     1)
26 X_val = torch.tensor(X_val, dtype=torch.float32)
27 y_val = torch.tensor(y_val, dtype=torch.float32).view(-1, 1)
28 X_test = torch.tensor(X_test, dtype=torch.float32)
29 y_test = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)
```

### RBF Layer Implementation

At first, I defined the RBF layer. This layer uses a set of trainable centers to compute distances between the input data and these centers, applying a Gaussian function to transform the distances into feature representations.

```

1 # Define RBF Layer
2 class RBFLayer(nn.Module):
3     def __init__(self, input_dim, num_centers, gamma=1.0):
4         super(RBFLayer, self).__init__()
5         self.centers = nn.Parameter(torch.rand(num_centers,
6         input_dim)) # Trainable centers
7         self.gamma = gamma
8
9     def forward(self, x):
10         diff = x.unsqueeze(1) - self.centers # Calculate
11         distance to centers
12         dist = torch.sum(diff ** 2, dim=-1)
13         return torch.exp(-self.gamma * dist) # Apply Gaussian
14         kernel

```

## RBF-Based Model Implementation

Then I defined a model consisting of one RBF layer followed by a Dense layer.

```

1 # Define RBF Model
2 class RBFModel(nn.Module):
3     def __init__(self, input_dim, num_centers):
4         super(RBFModel, self).__init__()
5         self.rbf = RBFLayer(input_dim, num_centers)
6         self.fc = nn.Linear(num_centers, 1)
7
8     def forward(self, x):
9         x = self.rbf(x)
10        x = self.fc(x)
11        return x

```

## Dense Neural Network Implementation

Then defined the Dense neural network model, which consists of three fully connected layers with ReLU activation functions between them:

```
1 # Define Dense Model
2 class DenseModel(nn.Module):
3     def __init__(self, input_dim, hidden_units):
4         super(DenseModel, self).__init__()
5         self.fc1 = nn.Linear(input_dim, hidden_units)
6         self.relu = nn.ReLU()
7         self.fc2 = nn.Linear(hidden_units, hidden_units)
8         self.fc3 = nn.Linear(hidden_units, 1)
9
10    def forward(self, x):
11        x = self.fc1(x)
12        x = self.relu(x)
13        x = self.fc2(x)
14        x = self.relu(x)
15        x = self.fc3(x)
16        return x
```

## Training Function

I implemented two essential functions used in training and evaluating machine learning models.

```
1 # Training function
2 def train_model(model, optimizer, criterion, X_train, y_train,
3                 X_val, y_val, epochs=200):
4     train_losses, val_losses = [], []
5     for epoch in range(epochs):
6         # Training phase
7         model.train()
8         optimizer.zero_grad()
9         y_pred = model(X_train)
10        loss = criterion(y_pred, y_train)
11        loss.backward()
12        optimizer.step()
13        train_losses.append(loss.item())
14
15        # Validation phase
16        model.eval()
17        with torch.no_grad():
18            y_val_pred = model(X_val)
19            val_loss = criterion(y_val_pred, y_val)
20            val_losses.append(val_loss.item())
```

```

20
21     return train_losses, val_losses

```

## Evaluation Function

```

1 # Evaluation function
2 def evaluate_model(model, X_test, y_test, criterion):
3     model.eval()
4     with torch.no_grad():
5         y_pred = model(X_test)
6         test_loss = criterion(y_pred, y_test)
7         r2 = r2_score(y_test.numpy(), y_pred.numpy())
8     return test_loss.item(), r2

```

## RBF Model Implementation

```

1 # Hyperparameters for RBF model
2 input_dim = X_train.shape[1]
3 num_centers = 64
4 gamma = 1
5 lr = 0.01
6 epochs = 200
7
8 # Initialize model, optimizer, and loss function
9 rbf_model = RBFModel(input_dim, num_centers)
10 optimizer = optim.Adam(rbf_model.parameters(), lr=lr)
11 criterion = nn.MSELoss()
12
13 # Train the RBF model
14 rbf_train_losses, rbf_val_losses = train_model(
15     rbf_model, optimizer, criterion, X_train, y_train, X_val,
16     y_val, epochs=epochs
17 )
18
19 # Evaluate the RBF model
20 rbf_test_loss, rbf_test_r2 = evaluate_model(rbf_model, X_test,
21     y_test, criterion)
22 print(f"RBF Model Test Loss: {rbf_test_loss:.4f}, R Score: {
23     rbf_test_r2:.4f}")

```

## Dense Model Implementation



```

1 # Hyperparameters for Dense model
2 hidden_units = 64
3
4 # Initialize model, optimizer, and loss function
5 dense_model = DenseModel(input_dim, hidden_units)
6 optimizer = optim.Adam(dense_model.parameters(), lr=lr)
7 criterion = nn.MSELoss()
8
9 # Train the Dense model
10 dense_train_losses, dense_val_losses = train_model(
11     dense_model, optimizer, criterion, X_train, y_train, X_val,
12     y_val, epochs=epochs
13 )
14 # Evaluate the Dense model
15 dense_test_loss, dense_test_r2 = evaluate_model(dense_model,
16     X_test, y_test, criterion)
17 print(f"Dense Model Test Loss: {dense_test_loss:.4f}, R Score
18       : {dense_test_r2:.4f}")

```

## Model Performance Results

The following table summarizes the performance metrics for the RBF and Dense models:

Model	Test MSE	Test $R^2$	Avg Train Loss	Avg Validation Loss	Train $R^2$	Validation $R^2$
RBF	0.6772	0.4932	1.7415	1.7987	0.5150	0.5277
Dense	0.3025	0.7736	0.4725	0.4588	0.7949	0.7815

Table 6: Performance Metrics for RBF and Dense Models.

From the results in Table 6, it can be observed that the Dense model significantly outperformed the RBF model across all metrics. The Dense model achieved a lower Test Mean Squared Error (MSE) of 0.3025 compared to 0.6772 for the RBF model, indicating better predictive accuracy. Similarly, the Dense model achieved a higher  $R^2$  score of 0.7736, reflecting a better fit to the data, whereas the RBF model obtained an  $R^2$  score of 0.4932. Furthermore, the Dense model demonstrated superior training and validation performance, with lower average training loss (0.4725) and validation loss (0.4588), compared to the RBF model's training loss (1.7415) and validation loss (1.7987). The Dense model also achieved consistently higher  $R^2$  scores for both training (0.7949) and validation (0.7815), highlighting its ability to capture the underlying patterns in the data more effectively than the RBF model. These results suggest that the Dense model provides better generalization and predictive performance on the given dataset.

## RBF Model Loss

The RBF model shows a gradual and steady decline in both training and validation losses over the 200 epochs. The convergence is evident as the loss curves stabilize towards the end, indicating that the model has reached its optimal parameters within the given training duration. However, the relatively high final loss values for both training and validation suggest that the RBF model might be underfitting the data. The minimal gap between training and validation loss curves indicates no significant overfitting, but it also implies that the model is not sufficiently complex to capture the intricate patterns in the dataset. Enhancing the RBF model's capacity, such as increasing the number of centers or experimenting with different kernel functions, might improve its performance.

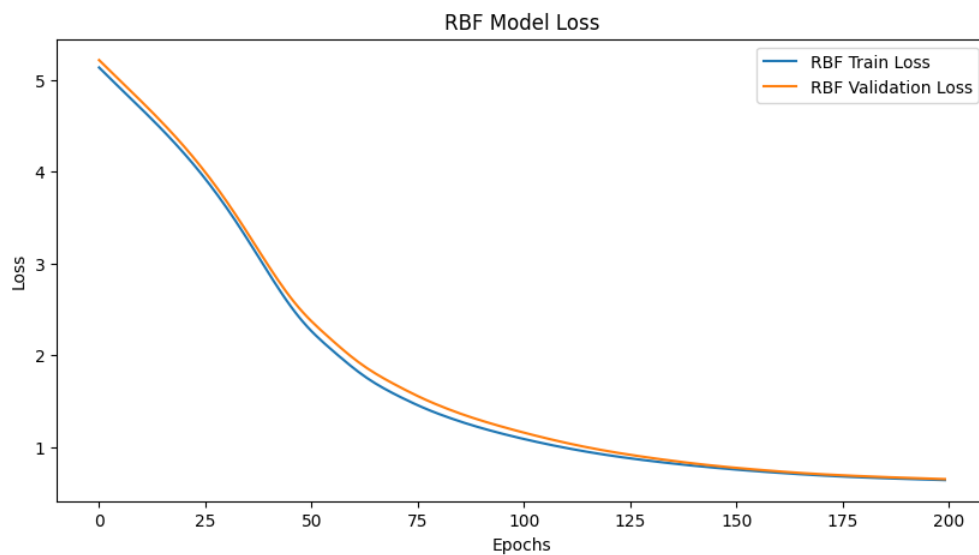


Figure 20: RBF Model Loss: Training and Validation

## Dense Model Loss

The Dense model exhibits a rapid decline in both training and validation losses during the initial epochs, which indicates that the model is effectively learning and adjusting its parameters to fit the data. The convergence of the loss curves towards the later epochs, as seen in their stabilization, suggests that the model is reaching its optimal configuration. The relatively low final loss values for both training and validation indicate that the model fits the data well. Furthermore, the negligible gap between the training and validation loss curves implies that the model is not overfitting, achieving a good generalization to unseen data. However, the consistent convergence at relatively low loss values suggests that the model has captured the dataset's patterns effectively. This indicates that the architecture of the Dense model is appropriate for the given problem, with no immediate need for additional complexity or regularization.

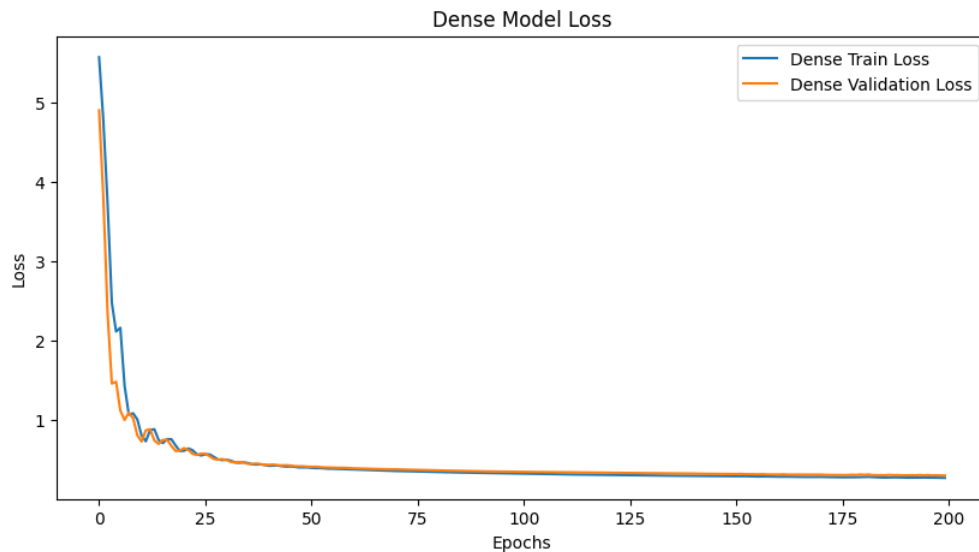


Figure 21: Dense Model Loss: Training and Validation