به نام خدا

محمدمهدی کرمی – ۴۰۰۰۸۳۷۳

لینک گیتهاب

سوال اول - بخش اول

لینک کولب سوال اول

در یک مسئله‌ی **طبقه‌بندی دوکلاسه**، استفاده از **ReLU در لایه‌ی ماقبل خروجی** و **سیگموید در لایه‌ی خروجی** می‌تواند مشکلاتی ایجاد کند. این مشکلات به چند دلیل اتفاق می‌افتند:

1. احتمال تولید مقادیر نادرست در لایه‌ی مخفی (ReLU)

- تابع **ReLU** مقدار ورودی‌های منفی را به صفر تبدیل می‌کند، در حالی که مقدار ورودی‌های مثبت را بدون تغییر نگه می‌دارد.
- این رفتار می‌تواند باعث شود که بعضی از نورون‌ها در **لایه‌ی ماقبل خروجی** مقدار **صفر** دریافت کرده و غیر فعال شوند.
- در نتیجه، در صورتی که مقدار زیادی از نورون‌های این لایه غیرفعال شوند، یادگیری مدل دچار مشکل می‌شود (**مشکل مرگ نورون‌ها** یا **Dying ReLU Problem**).

2. تأثیر بر توزیع ورودی به تابع سیگموید

- سیگموید یک **تابع غیرفعال‌کننده** است که خروجی را در بازه‌ی **(0,1)** فشرده می‌کند.
- خروجی‌های بزرگ **ReLU** می‌توانند باعث شوند که مقدار ورودی به سیگموید بسیار **بزرگ** یا **کوچک** باشد، که در نتیجه مقدار گرادیان در این نواحی تقریباً **صفر** خواهد شد (**مشکل ناپدید شدن گرادیان** یا **Vanishing Gradient**).
- این اتفاق باعث **کند شدن یادگیری مدل** و **عدم به‌روزرسانی مناسب وزن‌ها** می‌شود.

3. مشکل عدم تطابق توابع هزینه و خروجی شبکه

- در مسائل **طبقه‌بندی دوکلاسه**، معمولاً از **تابع هزینه‌ی باینری کراس انتروپی (Binary Cross-Entropy, BCE)** استفاده می‌شود.
- تابع BCE انتظار دارد که ورودی آن توزیع احتمال باشد (یعنی مقدار بین ۰ و ۱ داشته باشد).
- خروجی ReLU قبل از سیگموید می‌تواند مقادیر **بزرگ‌تر از ۱** را ایجاد کند که باعث می‌شود مدل خروجی‌های نادرستی تولید کند.

راه‌حل پیشنهادی

به جای **ReLU در لایه‌ی ماقبل خروجی**، می‌توان از یکی از روش‌های زیر استفاده کرد:

- **استفاده از توابع دیگر مانند tanh** به‌عنوان تابع فعال‌سازی لایه‌ی ماقبل خروجی، زیرا مقادیر خروجی را در بازه‌ی [-1,1][−1,1] نگه می‌دارد و رفتار بهتری نسبت به ReLU در این مرحله دارد.
- **حذف لایه‌ی ReLU و استفاده‌ی مستقیم از سیگموید در لایه‌ی خروجی**، زیرا سیگموید به‌تنهایی برای طبقه‌بندی دوکلاسه کافی است.

نتیجه‌گیری

استفاده از **ReLU در لایه‌ی ماقبل خروجی** و **سیگموید در لایه‌ی خروجی** می‌تواند منجر به مشکلاتی مانند **مرگ نورون‌ها، ناپدید شدن گرادیان و عدم تطابق توابع فعال‌سازی با تابع هزینه** شود. بنابراین، بهتر است از **سیگموید به‌تنهایی یا توابع فعال‌سازی مناسب‌تری مانند tanh** استفاده شود.

سوال اول - بخش دوم

تابع **Exponential Linear Unit (ELU)** به‌عنوان جایگزینی برای **ReLU** معرفی شده است و به‌صورت زیر تعریف می‌شود:

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

که در آن، **α** یک مقدار **مثبت** و **قابل تنظیم** است.

محاسبه گرادیان ELU

برای مشتق‌گیری از تابع ELU، دو حالت را بررسی می‌کنیم:

For $x \geq 0$:

$$\frac{d}{dx} ELU(x) = \frac{d}{dx} x = 1$$

For $x < 0$:

$$\frac{d}{dx} ELU(x) = \frac{d}{dx} \left(\alpha(e^x - 1)\right) = \alpha e^x$$

بنابراین، گرادیان تابع **ELU** برابر است با:

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

مزیت ELU نسبت به ReLU

یکی از مهم‌ترین مزایای **ELU** نسبت به **ReLU** این است که **مشکل مرگ نورون‌ها (Dying ReLU Problem)** را کاهش می‌دهد.

- در **ReLU**، اگر ورودی یک نورون منفی باشد، مقدار خروجی و گرادیان آن صفر می‌شود، که باعث می‌شود نورون **برای همیشه غیرفعال شود** و دیگر در یادگیری مشارکت نکند.
- در **ELU**، برای مقادیر منفی **خروجی صفر نمی‌شود** بلکه به مقدار کوچک **α(e^x - 1)** همگرا می‌شود. این باعث می‌شود که نورون‌ها همچنان مقدار گرادیان غیرصفر داشته باشند و در یادگیری مشارکت کنند.

نتیجه‌گیری

ELU به دلیل تولید **گرادیان غیرصفر در نواحی منفی** و جلوگیری از **مرگ نورون‌ها**، عملکرد بهتری نسبت به **ReLU** دارد، به‌ویژه در شبکه‌های عمیق.

سوال اول - بخش سوم

در این مسئله از نورون McCulloch-Pitts برای جداسازی نقاط داخل مثلث از نقاط خارج آن استفاده می‌شود. برای انجام این کار، ابتدا باید معادله هر یک از خطوط مرزی مثلث را پیدا کرده و سپس آن‌ها را به نورون‌ها منتقل کنیم.

برای هر یک از خطوط مثلث یک نورون ایجاد می‌کنیم که وزن‌ها و بایاس‌ها را از معادلات خطوط به دست می‌آورد. به این ترتیب، با استفاده از سه نورون، می‌توانیم وضعیت هر نقطه نسبت به سه خط مرزی مثلث را بررسی کنیم.

برای ترکیب خروجی این سه نورون و تشخیص اینکه نقطه داخل مثلث است یا نه، از یک نورون دیگر استفاده می‌کنیم که به ورودی‌های نورون‌های قبلی (خروجی‌های خطوط مرزی) یک عمل AND را اعمال می‌کند. نتیجه این نورون در نهایت نشان‌دهنده این است که آیا نقطه داخل مثلث است یا خیر.

کد نوشته شده برای حل این مسئله به شرح زیر عمل می‌کند:

۱. **تعریف نورون :McCulloch-Pitts** یک کلاس به نام McCulloch_Pitts_neuron ایجاد شده که دارای وزن‌ها و آستانه است و خروجی آن بر اساس این پارامترها محاسبه می‌شود.

۲. **تعریف خطوط مثلث:** برای هر دو نقطه از هر خط مثلث، یک نورون ایجاد شده که وظیفه دارد بررسی کند آیا نقطه داده شده در یک سمت خط قرار دارد یا خیر.

۳. **ترکیب خروجی‌ها:** یک نورون چهارم برای ترکیب خروجی‌های سه نورون قبلی و تصمیم‌گیری نهایی)با استفاده از عمل (AND ایجاد شده است.

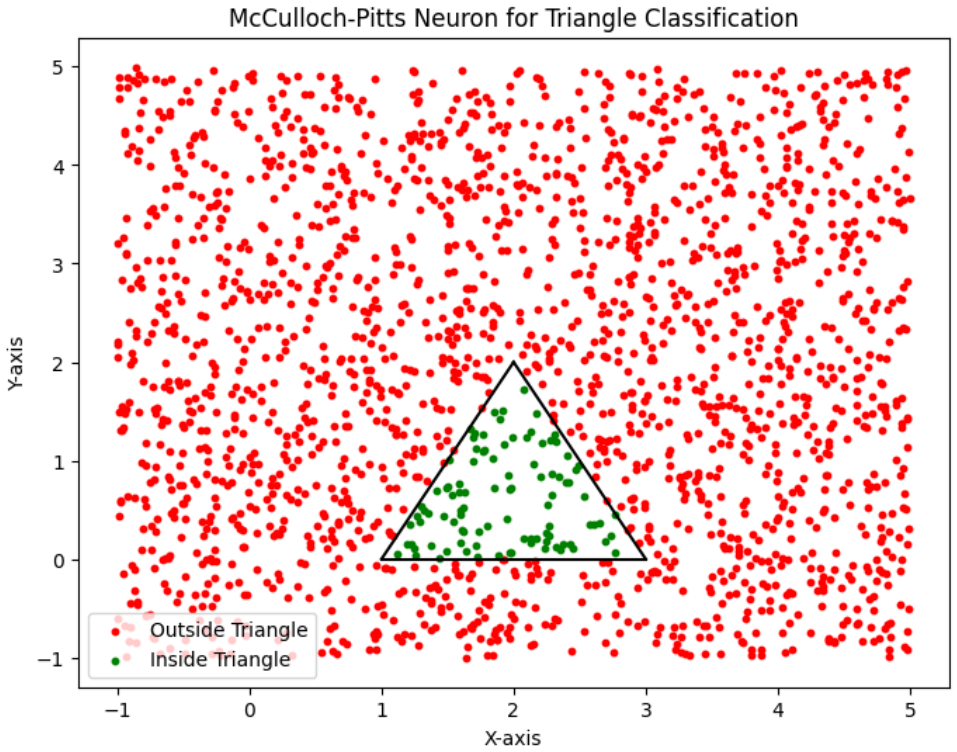
٤. **تولید داده‌های تصادفی:** ۲۰۰۰ نقطه تصادفی تولید می‌شود که هرکدام برای تعیین اینکه داخل مثلث هستند یا خیر به نورون‌ها وارد می‌شوند.

٥. **ترسیم نتایج:** نقاط داخل مثلث با رنگ سبز و خارج از مثلث با رنگ قرمز نمایش داده می‌شوند.

در نتیجه، خروجی نهایی یک نمودار است که نقاط داخل مثلث را با رنگ سبز و نقاط خارج از مثلث را با رنگ قرمز نشان می‌دهد.

این روش و کد می‌تواند برای مدل‌های ساده‌تر یا موارد مشابه نیز استفاده شود. در اینجا از نورون‌های McCulloch-Pitts استفاده کرده‌ایم که به دلیل سادگی و توانایی در تصمیم‌گیری‌های خطی برای مسائل مشابه مناسب هستند.

کد شما برای تولید نقاط تصادفی، استفاده از نورون‌ها برای تعیین وضعیت نقاط و ترسیم نمودار به خوبی نتیجه‌ای که می‌خواهید را نشان می‌دهد.



Total Points Generated: 2000

Points Inside Triangle: 98

Points Outside Triangle: 1902

Percentage Inside: 4.9 %

Percentage Outside: 95.1 %

سوال دوم - بخش اول

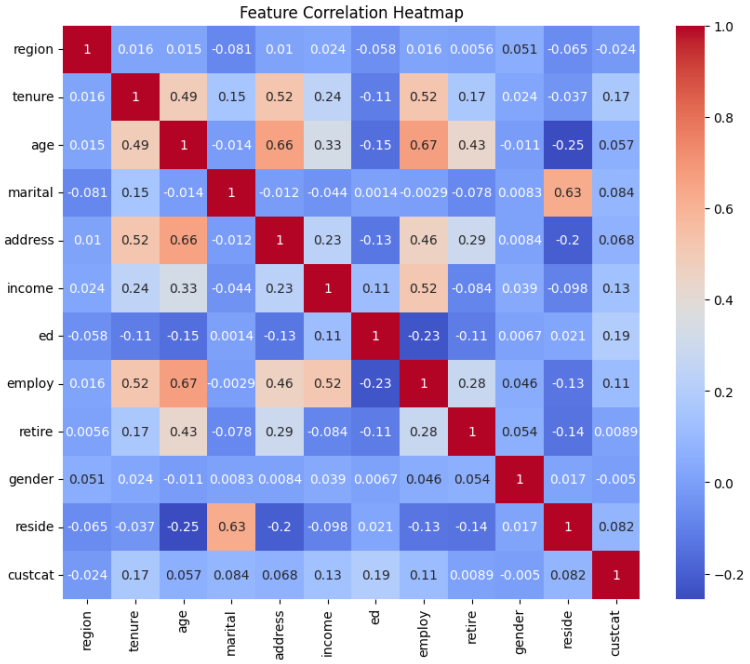
لینک کولب سوال دوم

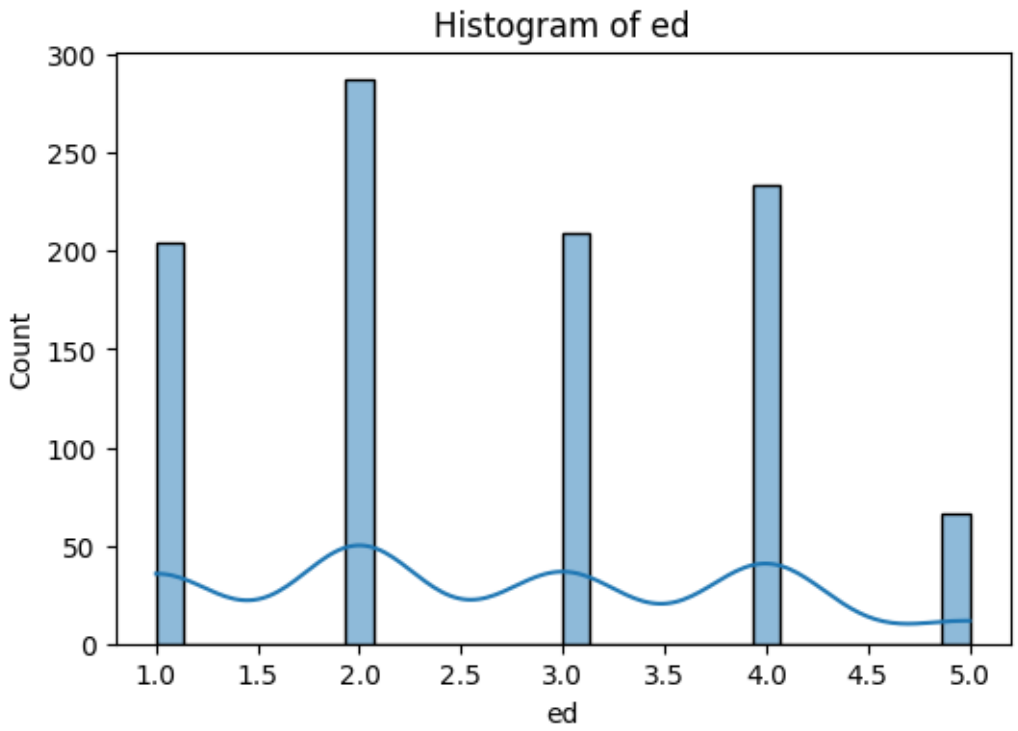
```
import pandas as pd

# Load dataset
file_path = "teleCust1000t.csv"
data = pd.read_csv(file_path)
```

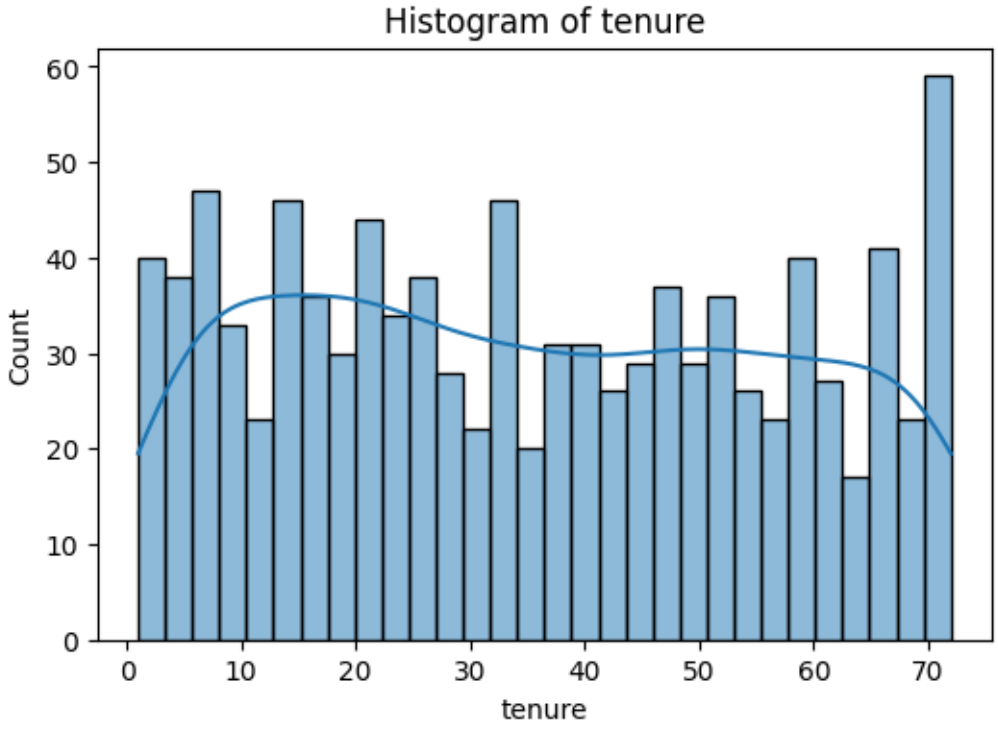
سوال دوم - بخش دوم

```
import seaborn as sns
import matplotlib.pyplot as plt

# Compute correlation matrix
corr_matrix = data.corr()

# Heatmap
```

```
plt.figure(figsize=(10,8))
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm")
plt.title("Feature Correlation Heatmap")
plt.show()

# Select the two most correlated features
correlated_features =
corr_matrix["custcat"].abs().sort_values(ascending=False).index[1:3]

# Plot histograms
for feature in correlated_features:
    plt.figure(figsize=(6, 4))
    sns.histplot(data[feature], kde=True, bins=30)
    plt.title(f"Histogram of {feature}")
    plt.show()
```







سوال دوم - بخش سوم

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
```

```
import tensorflow as tf

# Separate features and target
X = data.drop(columns=['custcat']).values
y = data['custcat'].values - data['custcat'].min()

# Normalize features
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# Train-test-validation split
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_full,
y_train_full, test_size=0.2, random_state=42)

# Convert labels to categorical format
num_classes = len(set(y))
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_val = tf.keras.utils.to_categorical(y_val, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)
```

سوال دوم - بخش چهارم

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

def build_model(layers, optimizer, dropout_rate=0.0, l2_reg=0):
    model = Sequential()
    model.add(Dense(layers[0], activation='relu',
input_shape=(X_train.shape[1],), kernel_regularizer=l2(l2_reg)))
    model.add(BatchNormalization())
    model.add(Dropout(dropout_rate))

    for units in layers[1:]:
        model.add(Dense(units, activation='relu',
kernel_regularizer=l2(l2_reg)))
        model.add(BatchNormalization())
        model.add(Dropout(dropout_rate))

    model.add(Dense(num_classes, activation='softmax'))
```

```
    model.compile(optimizer=optimizer, loss='categorical_crossentropy',
metrics=['accuracy'])

    return model

models = {}
histories = {}

configs = [
    {"layers": [50], "opt": SGD(learning_rate=0.01), "dropout": 0, "l2":
0},
    {"layers": [50], "opt": SGD(learning_rate=0.01), "dropout": 0.3, "l2":
0},
    {"layers": [50], "opt": SGD(learning_rate=0.01), "dropout": 0.3, "l2":
0.0001},
    {"layers": [100, 50], "opt": SGD(learning_rate=0.01), "dropout": 0.3,
"l2": 0.0001},
    {"layers": [100, 50], "opt": Adam(learning_rate=0.001), "dropout":
0.3, "l2": 0.0001},
    {"layers": [100, 50], "opt": RMSprop(learning_rate=0.001), "dropout":
0.3, "l2": 0.0001}
]

for config in configs:
    key =
f"{config['opt'].__class__.__name__}_L{config['layers']}_D{config['dropout
']}_L2{config['l2']}"
    print(f"Training {key}")

    model = build_model(config["layers"], config["opt"],
config["dropout"], config["l2"])
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val),
epochs=50, batch_size=64, verbose=1)

    models[key] = model
    histories[key] = history

    test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
    print(f"{key} - Test Accuracy: {test_acc:.2f}, Test Loss:
{test_loss:.2f}")

    # Plot Loss
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
    plt.title(f'Loss - {key}')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

نتایج خروجی کد بالا

```
========== ONE_LAYER MODEL EXPERIMENTS ==========
Testing neurons configuration: [50]
 -> Validation Accuracy: 0.3875
Testing neurons configuration: [100]
 -> Validation Accuracy: 0.4187
Best neurons configuration: [100] with Val Acc: 0.4187
Testing Batch Normalization = False
 -> Validation Accuracy: 0.3500
Testing Batch Normalization = True
 -> Validation Accuracy: 0.4563
Best Batch Normalization: True with Val Acc: 0.4563
Testing Dropout = 0
 -> Validation Accuracy: 0.4563
Testing Dropout = 0.3
 -> Validation Accuracy: 0.4250
Best Dropout: 0 with Val Acc: 0.4563
Testing L2 Regularization = 0
 -> Validation Accuracy: 0.4250
Testing L2 Regularization = 0.0001
 -> Validation Accuracy: 0.4437
Best L2 Regularization: 0.0001 with Val Acc: 0.4437
Testing Optimizer: SGD
 -> Validation Accuracy: 0.4437
Testing Optimizer: Adam
 -> Validation Accuracy: 0.4812
Testing Optimizer: RMSprop
 -> Validation Accuracy: 0.4812
Best Optimizer: Adam with Val Acc: 0.4812
Training final best model for this architecture ...
Final Test Accuracy for one_layer model: 0.3700

========== TWO_LAYERS MODEL EXPERIMENTS ==========
Testing neurons configuration: [100, 50]
 -> Validation Accuracy: 0.4062
Testing neurons configuration: [150, 100]
 -> Validation Accuracy: 0.3750
Best neurons configuration: [100, 50] with Val Acc: 0.4062
Testing Batch Normalization = False
 -> Validation Accuracy: 0.3750
Testing Batch Normalization = True
 -> Validation Accuracy: 0.4500
Best Batch Normalization: True with Val Acc: 0.4500
Testing Dropout = 0
 -> Validation Accuracy: 0.4500
Testing Dropout = 0.3
 -> Validation Accuracy: 0.4313
Best Dropout: 0 with Val Acc: 0.4500
```

```
Testing L2 Regularization = 0
 -> Validation Accuracy: 0.4500
Testing L2 Regularization = 0.0001
 -> Validation Accuracy: 0.4437
Best L2 Regularization: 0 with Val Acc: 0.4500
Testing Optimizer: SGD
 -> Validation Accuracy: 0.4500
Testing Optimizer: Adam
 -> Validation Accuracy: 0.4437
Testing Optimizer: RMSprop
 -> Validation Accuracy: 0.4500
Best Optimizer: SGD with Val Acc: 0.4500
Training final best model for this architecture ...
Final Test Accuracy for two_layers model: 0.3650
```

بهترین مدل برای هر دو مدل یک لایه و دولایه را پیدا کردیم؛ در بخش بعد نمودار آموزش و آزمایش آن‌ها را رسم می‌کنیم.

سوال دوم - بخش پنجم

```
import numpy as np

best_model_key = max(models, key=lambda k: models[k].evaluate(X_test,
y_test, verbose=0)[1])
best_model = models[best_model_key]

print(f"Best Model: {best_model_key}")

# Test Accuracy
test_loss, test_acc = best_model.evaluate(X_test, y_test, verbose=0)
print(f"Best Model Test Accuracy: {test_acc:.2f}, Test Loss:
{test_loss:.2f}")

# Random Predictions
random_indices = np.random.choice(X_test.shape[0], 10, replace=False)
random_inputs = X_test[random_indices]
actual_labels = y_test[random_indices]
predicted_probs = best_model.predict(random_inputs)
predicted_labels = np.argmax(predicted_probs, axis=1)

for i, index in enumerate(random_indices):
    print(f"Sample {i+1}: Actual: {np.argmax(actual_labels[i])},
Predicted: {predicted_labels[i]}")
```

در این قسمت دقت هر مدل را بررسی می‌کنیم، سپس برای ۱۰ نمونه آزمایش، خروجی مدل و خروجی صحیح را چاپ می‌کنیم. نتایج کد عبارت است از:

```
One-Layer Model Test Accuracy: 0.3700
Random 10 Test Samples - One-Layer Model:
Index Predicted  True
21    1          1
79    0          2
66    0          0
```

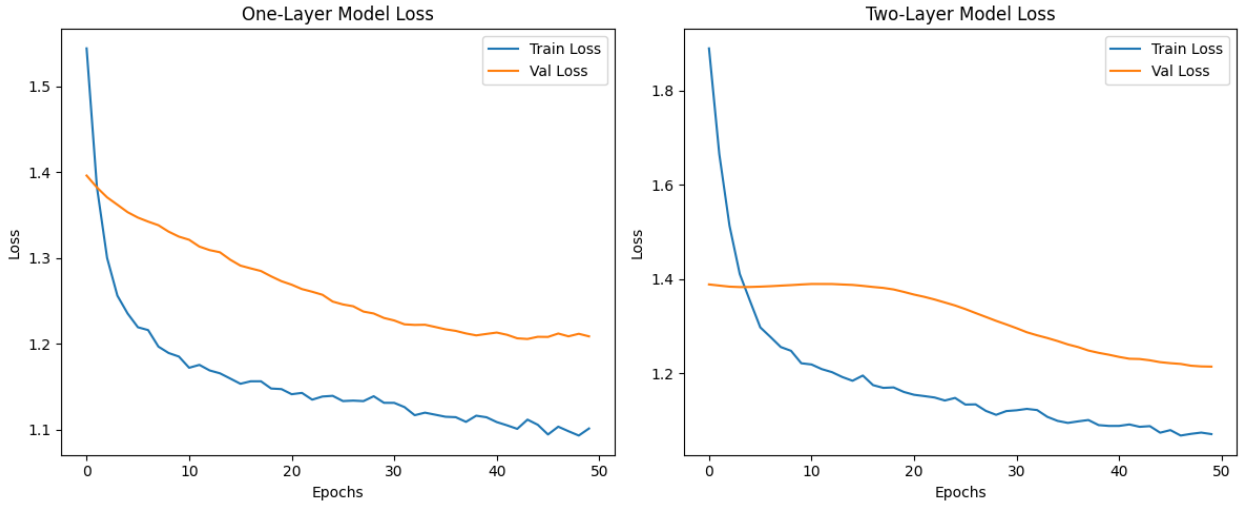
```
113   0           2
49    3           2
24    2           0
90    3           2
117   0           0
54    2           0
118   1           2

Two-Layer Model Test Accuracy: 0.3650
Random 10 Test Samples - Two-Layer Model:
Index Predicted  True
147   0           0
151   3           1
92    2           2
59    0           0
54    2           0
101   2           2
162   1           0
156   1           0
155   2           2
64    2           1
```

سوال دوم - بخش ششم

```python
ensemble_preds = np.zeros_like(y_test, dtype=np.float32)

for model in models.values():
    ensemble_preds += model.predict(X_test)

ensemble_preds /= len(models)
ensemble_labels = np.argmax(ensemble_preds, axis=1)

ensemble_accuracy = np.mean(np.argmax(y_test, axis=1) == ensemble_labels)
print(f"Ensemble Model - Test Accuracy: {ensemble_accuracy:.2f}")
```

نتایج کد:

```
========== ENSEMBLE OF THE TWO MODELS ==========
Ensemble Test Accuracy: 0.3800

--- Analysis ---
One-Layer Model Accuracy: 0.3700
Two-Layer Model Accuracy: 0.3650
Ensemble Model Accuracy: 0.3800
```



سوال سوم – بخش اول

لینک کولب سوال سوم

```
from PIL import Image, ImageDraw
import random

def convertImageToBinary(path):
    """
    Convert an image to a binary representation based on pixel intensity.

    Args:
        path (str): The file path to the input image.

    Returns:
        list: A binary representation of the image where white is
represented by -1 and black is represented by 1.
    """
    # Open the image file.
    image = Image.open(path)

    # Create a drawing tool for manipulating the image.
    draw = ImageDraw.Draw(image)

    # Determine the image's width and height in pixels.
    width = image.size[0]
    height = image.size[1]

    # Load pixel values for the image.
    pix = image.load()

    # Define a factor for intensity thresholding.
```

```
    factor = 100

    # Initialize an empty list to store the binary representation.
    binary_representation = []

    # Loop through all pixels in the image.
    for i in range(width):
        for j in range(height):
            # Extract the Red, Green, and Blue (RGB) values of the pixel.
            red = pix[i, j][0]
            green = pix[i, j][1]
            blue = pix[i, j][2]

            # Calculate the total intensity of the pixel.
            total_intensity = red + green + blue

            # Determine whether the pixel should be white or black based
on the intensity.
            if total_intensity > (((255 + factor) // 2) * 3):
                red, green, blue = 255, 255, 255  # White pixel
                binary_representation.append(-1)
            else:
                red, green, blue = 0, 0, 0  # Black pixel
                binary_representation.append(1)

            # Set the pixel color accordingly.
            draw.point((i, j), (red, green, blue))

    # Clean up the drawing tool.
    del draw

    # Return the binary representation of the image.
    return binary_representation
```

تابع اول ابتدا ابعاد تصویر ورودی (طول و عرض) را استخراج کرده و سپس یک مقدار آستانه (threshold) یا فاکتور (factor) برای تعیین سفید یا سیاه بودن پیکسل‌ها مشخص می‌کند. در ادامه، مقادیر RGB تمام پیکسل‌ها را دریافت کرده و شدت کلی (total intensity) را محاسبه می‌کند. سپس با مقایسه این مقدار با آستانه تعیین‌شده، پیکسل‌ها را به رنگ سفید یا سیاه تبدیل کرده و نتیجه را در تصویر باینری (binary_representation) ذخیره می‌کند.

در اینجا، ماتریس **binary_img** دارای همان ابعاد تصویر ورودی است، اما ابتدا به مقیاس خاکستری (gray scale) تبدیل شده و دیگر مقادیر RGB ندارد. در این فرآیند، پیکسل‌هایی که مقدار آن‌ها بیشتر از ۷۵ باشد، مقدار **True** و بقیه مقدار **False** دریافت می‌کنند. در نهایت، این مقادیر با استفاده از **astype(np.int8)** به اعداد ۰ و ۱ تبدیل می‌شوند.

در تابع دوم، برای افزودن نویز به تصویر، یک پارامتر **noise_factor** در نظر گرفته شده که میزان نویز را تعیین می‌کند. نویز به این صورت اعمال می‌شود که ابتدا یک مقدار تصادفی در بازه **noise_factor-** تا **noise_factor** تولید شده و سپس این مقدار به مقادیر RGB پیکسل‌های تصویر اضافه می‌شود.

سوال سوم – بخش دوم

```
from PIL import Image, ImageDraw
import random

def generate_noisy_images():
    """
    Generate noisy versions of a set of images and save them with new
filenames.
    """
    image_paths = [
        "/content/1.jpg",
        "/content/2.jpg",
        "/content/3.jpg",
        "/content/4.jpg",
        "/content/5.jpg"
    ]

    for i, image_path in enumerate(image_paths, start=1):
        noisy_image_path = f"/content/noisy_{i}.jpg"
        apply_noise_to_image(image_path, noisy_image_path)
        print(f"Noisy image saved: {noisy_image_path}")

def apply_noise_to_image(input_path, output_path, noise_factor=50):
    """
    Apply random noise to an image and save the modified version.

    Args:
        input_path (str): Path to the input image.
        output_path (str): Path to save the noisy image.
        noise_factor (int): Intensity of the noise added to the image.
    """
    # Open the input image
    image = Image.open(input_path)
    draw = ImageDraw.Draw(image)
    width, height = image.size
    pixels = image.load()

    for x in range(width):
        for y in range(height):
            noise = random.randint(-noise_factor, noise_factor)
            r, g, b = pixels[x, y]
```

```
            # Apply noise and keep values in the valid range (0-255)
            r = max(0, min(255, r + noise))
            g = max(0, min(255, g + noise))
            b = max(0, min(255, b + noise))

            draw.point((x, y), (r, g, b))

    # Save the modified image
    image.save(output_path, "JPEG")
    del draw  # Clean up

# Generate and save noisy images
generate_noisy_images()
```

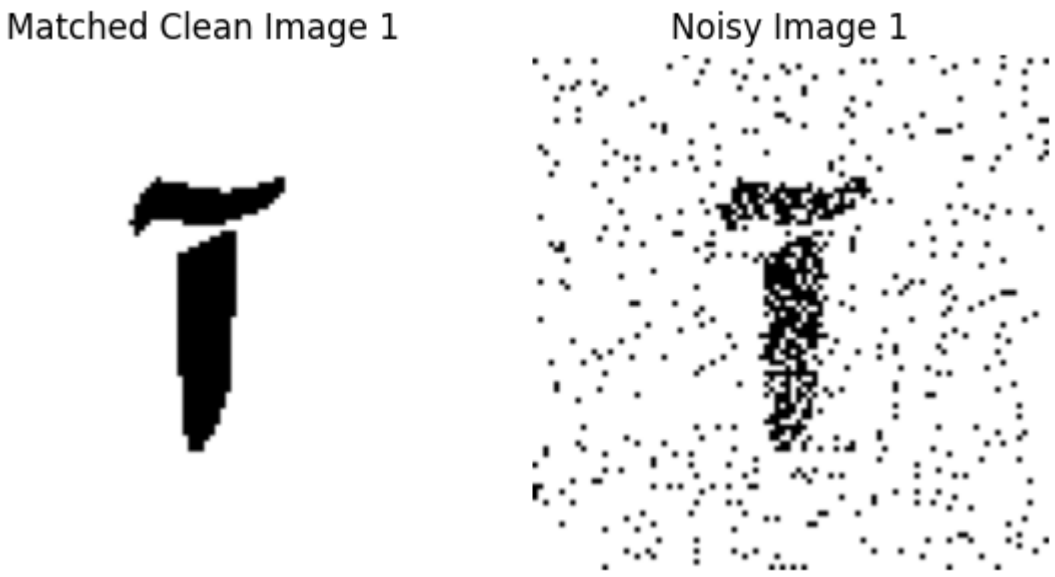
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

class HammingNetwork:
    def __init__(self, clean_images):
        """
        Initialize the Hamming Network with clean binary images.

        Parameters:
            clean_images: list of np.array, the set of normal binary
images.
        """
        self.clean_images = [image.flatten() for image in clean_images]  #
Flatten for vector processing

    def match(self, noisy_image):
        """
        Matches the noisy binary image to the closest clean image.

        Parameters:
            noisy_image: np.array, the noisy binary image to be matched.

        Returns:
            index: int, the index of the matched clean image.
            matched_image: PIL.Image.Image, the matched clean image.
        """
        noisy_image = noisy_image.flatten()  # Flatten the noisy image
        # Compute Hamming distances to each clean image
```

```
        distances = [np.sum(noisy_image != clean_image) for clean_image in
self.clean_images]
        index = np.argmin(distances)  # Find the index of the smallest
distance

        # Convert the matched image back to 2D array
        matched_array = self.clean_images[index].reshape(96, 96)

        # Convert the binary array (0s and 1s) back to a grayscale image
        matched_image = Image.fromarray((matched_array *
255).astype(np.uint8))  # Scale 0/1 to 0/255
        return index, matched_image

def load_and_preprocess_image(filepath, size=(96, 96)):
    """
    Loads and preprocesses an image for the Hamming Network.

    Parameters:
        filepath: str, path to the image file.
        size: tuple, dimensions to resize the image to.

    Returns:
        np.array: Binary representation of the image (0s and 1s).
    """
    # Load the image
    img = Image.open(filepath).convert("L")  # Convert to grayscale
    img = img.resize(size)  # Resize to 96x96
    # Convert to binary (thresholding)
    binary_img = np.array(img) > 75  # Threshold at 75
    return binary_img.astype(np.int8)  # Convert to 0s and 1s

# Paths to clean and noisy images
clean_image_paths = ["/content/1.jpg", "/content/2.jpg", "/content/3.jpg",
"/content/4.jpg", "/content/5.jpg"]
noisy_image_path = "noisy1.jpg"

# Load and preprocess images
clean_images = [load_and_preprocess_image(path) for path in
clean_image_paths]
noisy_image = load_and_preprocess_image(noisy_image_path)

# Initialize the Hamming Network
hamming_network = HammingNetwork(clean_images)

# Match the noisy image to a clean image
```

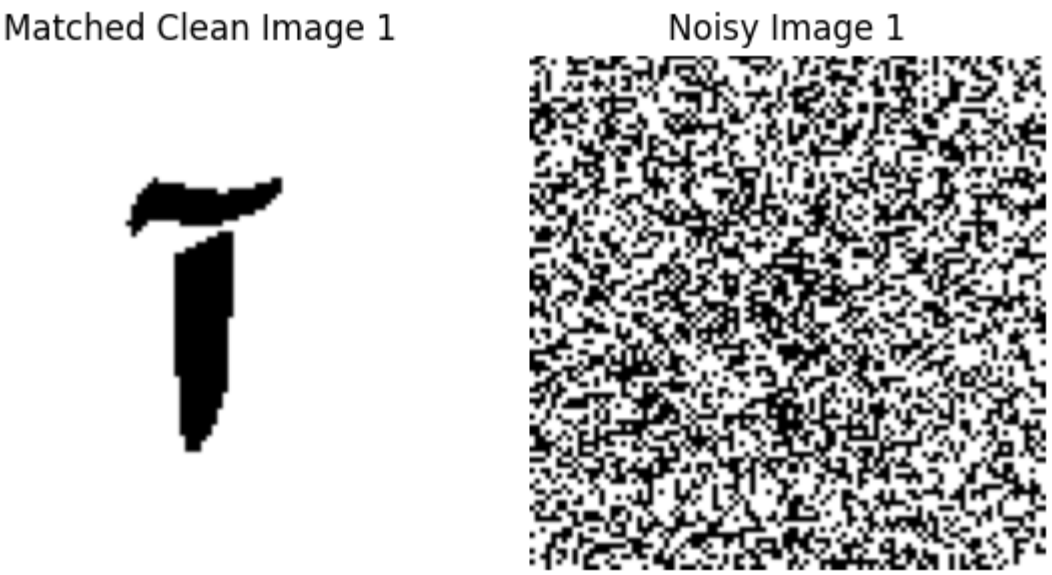
```
index, matched_image = hamming_network.match(noisy_image)

fig, axes = plt.subplots(1, 2, figsize=(6, 3))  # 1 row, 2 columns

axes[0].imshow(matched_image, cmap='gray')  # Display in grayscale
axes[0].set_title(f"Matched Clean Image {index + 1}")
axes[0].axis("off")  # Hide axes

axes[1].imshow(noisy_image, cmap='gray')  # Display in grayscale
axes[1].set_title(f"Noisy Image 1")
axes[1].axis("off")  # Hide axes
plt.tight_layout()
plt.show()
```

به دلیل سطح نویز اعمال‌شده، شبکه در برخی موارد خروجی نادرستی تولید می‌کند که نمونه‌ای از آن در تصاویر پایین مشاهده می‌شود.

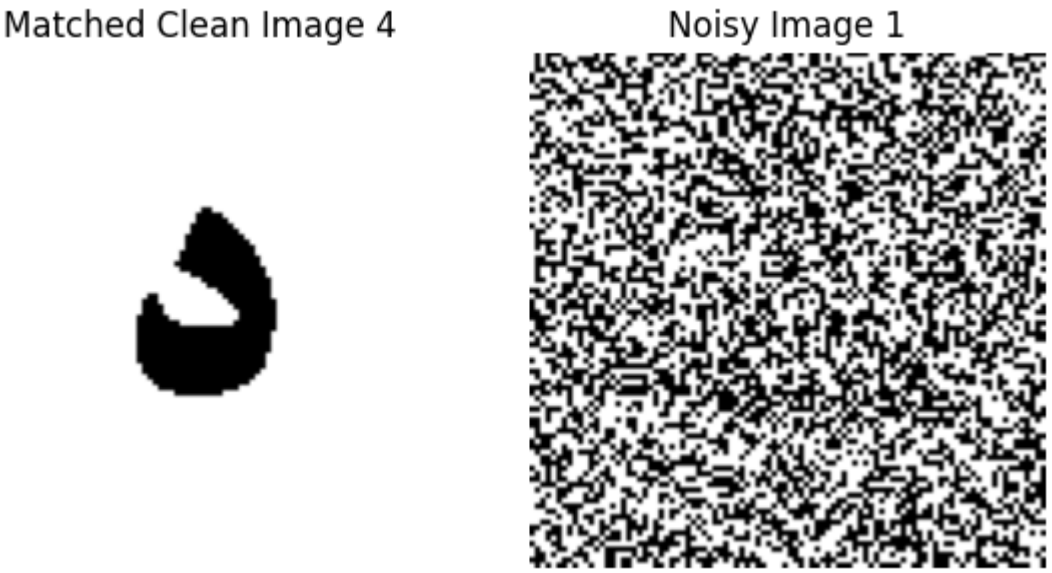
noise_factor = 200



noise_factor = 1000



noise_factor = 2000



سوال سوم – بخش سوم

```
def generateMissingPointImages():
    # List of image file paths
    image_paths = [
        "/content/1.jpg",
        "/content/2.jpg",
        "/content/3.jpg",
        "/content/4.jpg",
        "/content/5.jpg"
    ]
```

```
    for i, image_path in enumerate(image_paths, start=1):
        missing_point_path = f"/content/MissingPoint{i}.jpg"
        getNoisyBinaryImage(image_path, missing_point_path)
        print(f"Noisy image for {image_path} generated and saved as
{missing_point_path}")

def getNoisyBinaryImage(input_path, output_path):
    """
    Add noise to an image and save it as a new file.

    Args:
        input_path (str): The file path to the input image.
        output_path (str): The file path to save the noisy image.
    """
    # Open the input image.
    image = Image.open(input_path)

    # Create a drawing tool for manipulating the image.
    draw = ImageDraw.Draw(image)

    # Determine the image's width and height in pixels.
    width = image.size[0]
    height = image.size[1]

    # Load pixel values for the image.
    pix = image.load()

    # Define a factor for introducing noise.
    noise_factor = 500

    # Loop through all pixels in the image.
    for i in range(width):
        for j in range(height):
            # Generate a random noise value within the specified factor.
            rand = random.randint(0, noise_factor)

            # Add the noise to the Red, Green, and Blue (RGB) values of
the pixel.
            red = pix[i, j][0] + rand
            green = pix[i, j][1] + rand
            blue = pix[i, j][2] + rand

            # Ensure that RGB values stay within the valid range (0-255).
            if red < 0:
```

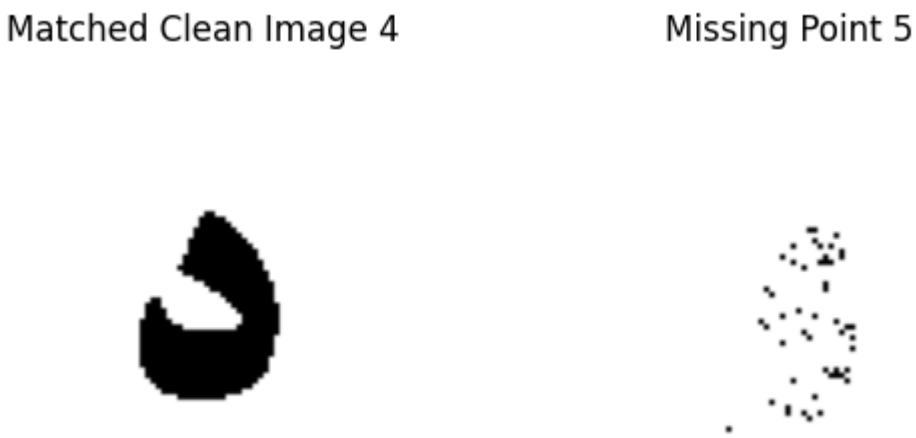
```
                red = 0
            if green < 0:
                green = 0
            if blue < 0:
                blue = 0

            # Set the pixel color accordingly.
            draw.point((i, j), (red, green, blue))

    # Save the noisy image as a file.
    image.save(output_path, "JPEG")

    # Clean up the drawing tool.
    del draw

# Generate noisy images and save them
generateMissingPointImages()
```

```
def load_and_preprocess_image(filepath, size=(96, 96)):
    """
    Loads and preprocesses an image for the Hamming Network.

    Parameters:
        filepath: str, path to the image file.
        size: tuple, dimensions to resize the image to.

    Returns:
        np.array: Binary representation of the image (0s and 1s).
    """
    # Load the image
    img = Image.open(filepath).convert("L")  # Convert to grayscale
    img = img.resize(size)  # Resize to 96x96
    # Convert to binary (thresholding)
    binary_img = np.array(img) > 75  # Threshold at 75
    return binary_img.astype(np.int8)  # Convert to 0s and 1s


# Paths to clean and noisy images
clean_image_paths = ["/content/1.jpg", "/content/2.jpg", "/content/3.jpg",
"/content/4.jpg", "/content/5.jpg"]
missing_point_path = "MissingPoint5.jpg"

# Load and preprocess images
clean_images = [load_and_preprocess_image(path) for path in
clean_image_paths]
```

```
missing_point_image = load_and_preprocess_image(missing_point_path)

# Initialize the Hamming Network
hamming_network = HammingNetwork(clean_images)

# Match the noisy image to a clean image
index, matched_image = hamming_network.match(missing_point_image)

fig, axes = plt.subplots(1, 2, figsize=(6, 3))  # 1 row, 2 columns

axes[0].imshow(matched_image, cmap='gray')  # Display in grayscale
axes[0].set_title(f"Matched Clean Image {index + 1}")
axes[0].axis("off")  # Hide axes

axes[1].imshow(missing_point_image, cmap='gray')  # Display in grayscale
axes[1].set_title(f"Missing Point 5")
axes[1].axis("off")  # Hide axes
plt.tight_layout()
plt.show()
```

برای بهبود عملکرد شبکه در تصاویر Missing Point، می‌توان چند نمونه از این تصاویر را برای آموزش مدل در نظر گرفت.

noise_factor = 1000



سوال چهارم

لینک کولب سوال چهارم

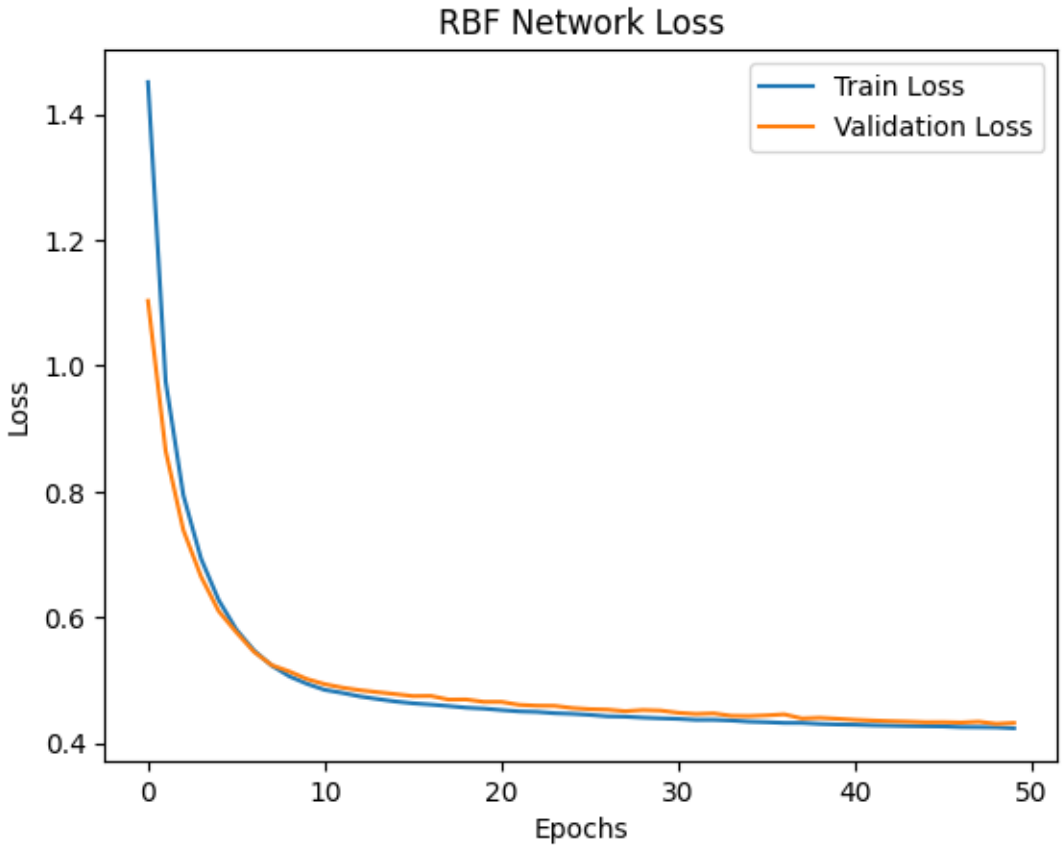
در این مسئله مشاهده می‌شود که مقدار **Loss** در مدل **Dense** کمتر از مقدار آن در مدل **RBF** است، که نشان‌دهنده عملکرد بهتر مدل **Dense** در پیش‌بینی مقدار هدف است. این تفاوت عملکرد را می‌توان به ویژگی‌های ذاتی این دو مدل نسبت داد.

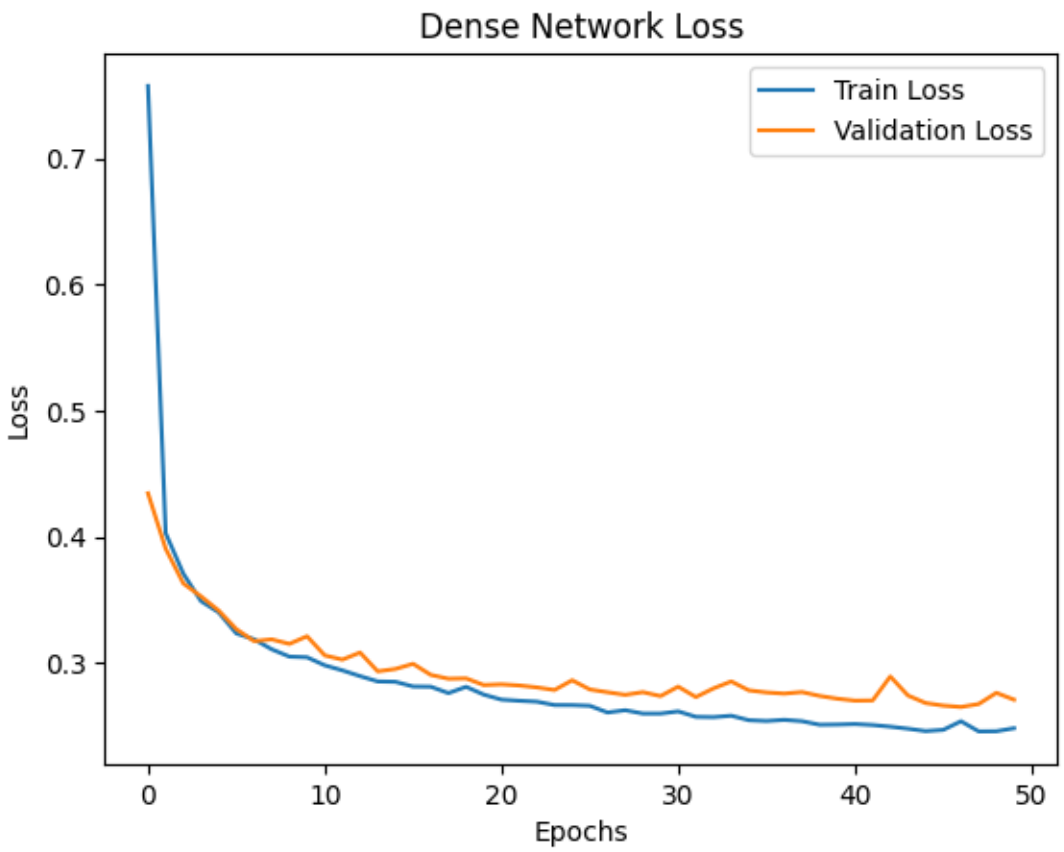
مدل **RBF** معمولاً زمانی عملکرد مطلوبی دارد که داده‌ها دارای توزیع خوشه‌ای باشند، یعنی بتوان آن‌ها را به گروه‌های مشخصی تقسیم کرد که هر گروه دارای ویژگی‌های مشابهی باشد. این مدل با استفاده از توابع پایه‌ای شعاعی، فاصله نمونه‌ها از مراکز مشخصی را محاسبه کرده و براساس آن پیش‌بینی انجام می‌دهد. در صورتی که داده‌ها به‌خوبی در قالب خوشه‌ها قرار نگیرند، این مدل نمی‌تواند روابط میان ورودی‌ها و خروجی‌ها را به‌درستی یاد بگیرد.

از طرف دیگر، مدل **Dense** (شبکه عصبی چندلایه) به دلیل داشتن چندین لایه‌ی متراکم با توابع فعال‌سازی مانند **ReLU** قادر است روابط پیچیده و غیرخطی بین متغیرهای ورودی و خروجی را بهتر مدل‌سازی کند. در این مسئله، به نظر می‌رسد که داده‌ها ساختار پیچیده و غیرخطی دارند و نمی‌توان آن‌ها را به سادگی در خوشه‌های مشخصی دسته‌بندی کرد. به همین دلیل، مدل **Dense** که توانایی یادگیری الگوهای غیرخطی را دارد، عملکرد بهتری در این پیش‌بینی ارائه داده است.

بنابراین، در مسائلی که داده‌ها دارای الگوهای توزیع پراکنده و روابط پیچیده هستند، استفاده از مدل‌های **Dense** که توانایی تقریب توابع غیرخطی را دارند، گزینه‌ی مناسب‌تری خواهد بود. در مقابل، مدل‌های **RBF** زمانی بهتر عمل می‌کنند که داده‌ها به صورت خوشه‌ای و دارای ساختار مشخص باشند.

```
RBF Network - Loss: 0.4319261610507965, MAE: 0.471592515707016
Dense Network - Loss: 0.270912766456604, MAE: 0.3544943034648895
```




Dense Network Loss
Train Loss
Validation Loss
Loss
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
Epochs

پایان