Aprendizagem 2023
Homework III – Group 023
(ist102848, ist103525)

Part I: Pen and paper

1. Consider the problem of learning a regression model from 4 bivariate observations {(0.7, -0.3), (0.4, 0.5), (0.2, 0.8), (-0.4, 0.3)} and their targets {0.8, 0.6, 0.3, 0.3}.

   (a) Given the radial basis function, $\phi_j(x) = e^{-\frac{\|x-c_j\|^2}{2}}$, that transforms the original space onto a new space characterized by the similarity of the original observations to the following data points:

   $$c_1 = (0, 0)$$
   $$c_2 = (1, -1)$$
   $$c_3 = (-1, 1)$$

   Learn the Ridge regression (l_2 regularization) using the closed solution with $\lambda = 0.1$.

   We begin by applying the radial basis function and data points to our observations, to get our transformed matrix:

   First we calculate our Φ matrix after the transformation

   $$\Phi = \begin{bmatrix} 1 & 0,748 & 0,748 & 0,101 \\ 1 & 0,815 & 0,271 & 0,331 \\ 1 & 0,772 & 0,963 & 0,772 \\ 1 & 0,882 & 0,161 & 0,654 \end{bmatrix}$$

   Loss function with regularization:

   Using Ridge regression

   $$E(w) = \frac{1}{2} \sum_{i=1}^{m} (z_i - w^T \cdot x_i)^2 + \frac{\lambda}{2} \|w\|^2$$

To obtain the best weights, we minimize the loss function by taking its derivative and finding where it equals zero. Since the loss function has positive coefficients, this zero point is a minimum.

To maximize w, we need to find the $E(w)$ minimum

$$\nabla E(w) = \nabla \left(\frac{1}{2} (z - x \cdot w)^T (z - x \cdot w) + \frac{\lambda}{2} w^T w \right) = 0$$

$$\Leftrightarrow -x^T z + x^T \cdot x \cdot w + \lambda \cdot w = 0$$

$$\Leftrightarrow x^T z = (x^T \cdot x + \lambda I) \cdot w$$

$$\Leftrightarrow w = (x^T \cdot x + \lambda I)^{-1} \cdot x^T \cdot z$$

After having developed the above expression we get to the closed form solution for Ridge Regression.

By inputting our transformed matrix into the above expression we learn the weights:

$$w = (\Phi^T \cdot \Phi + \lambda I)^{-1} \cdot \Phi^T \cdot z = \begin{bmatrix} 0,33974 \\ 0,19945 \\ 0,40096 \\ -0,29600 \end{bmatrix}$$

(b) Compute the training RMSE for the learnt regression.

After calculating the model's prediction for each observation we can compare them with the original output using the RMSE's formula.

b)

$$RMSE(\hat{z}, z) = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (z_i - \hat{z}_i)^2}$$

$$= \sqrt{\frac{1}{4} (0,04^2 + 0,09^2 + 0,01^2 + 0,09^2)}$$

$$= 0.06508$$

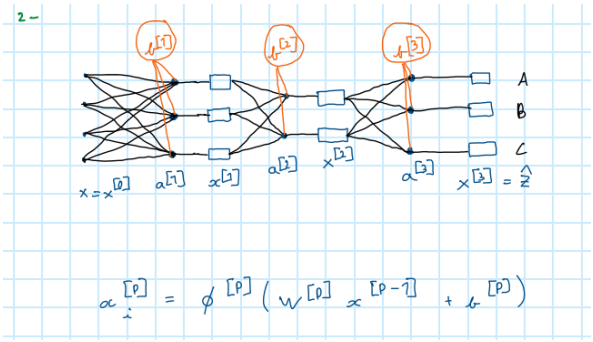
$$\hat{z}_i = w^T x_i$$

$$\hat{z}_0 = 0,75844$$
$$\hat{z}_1 = 0,51232$$
$$\hat{z}_2 = 0,30905$$
$$\hat{z}_3 = 0,38629$$

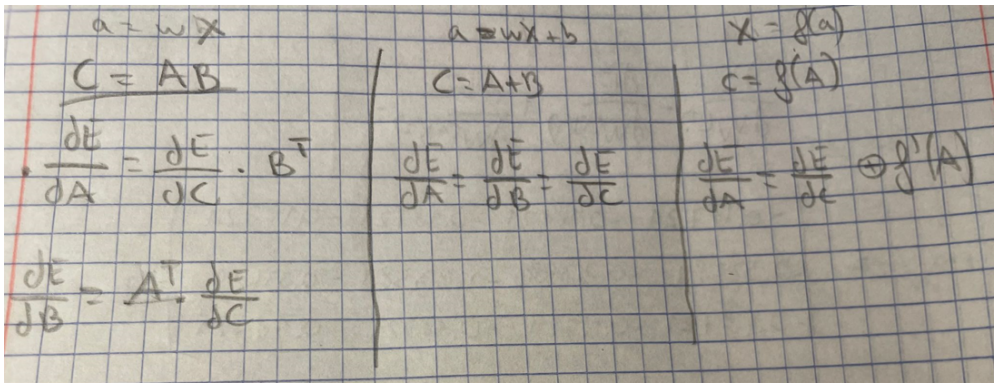
$$\hat{z}_i - z_i \approx (-0,04 \quad -0,09 \quad -0,01 \quad 0,09)$$

2. Consider a MLP classifier of three outcomes –A, B and C. Perform one batch gradient descent update (with learning rate $\eta = 0.1$) for training observations x_1 and x_2.

Before beginning, here is how the MLP looks like and the notation we chose to tackle this problem.

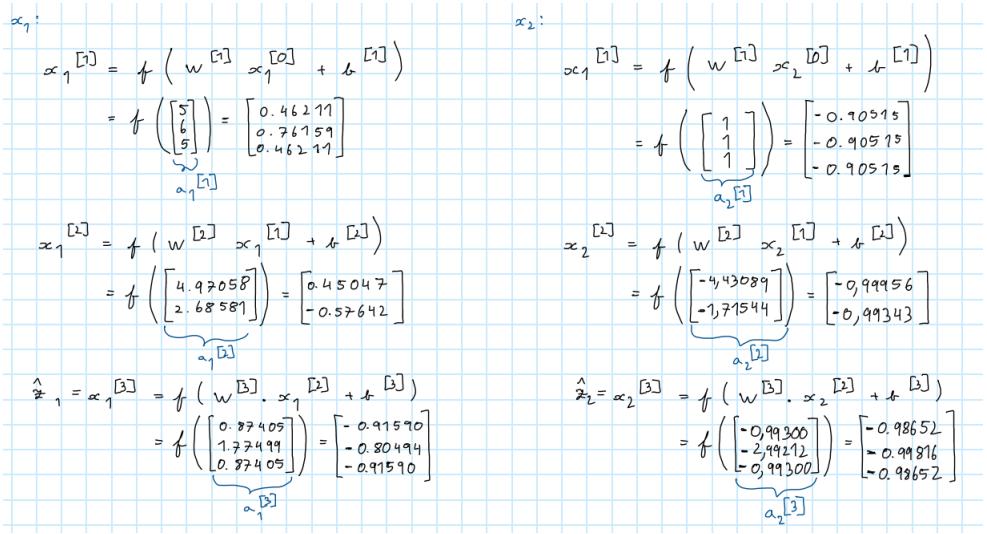


For the upcoming calculations, we are going to calculate many derivatives using these rules:



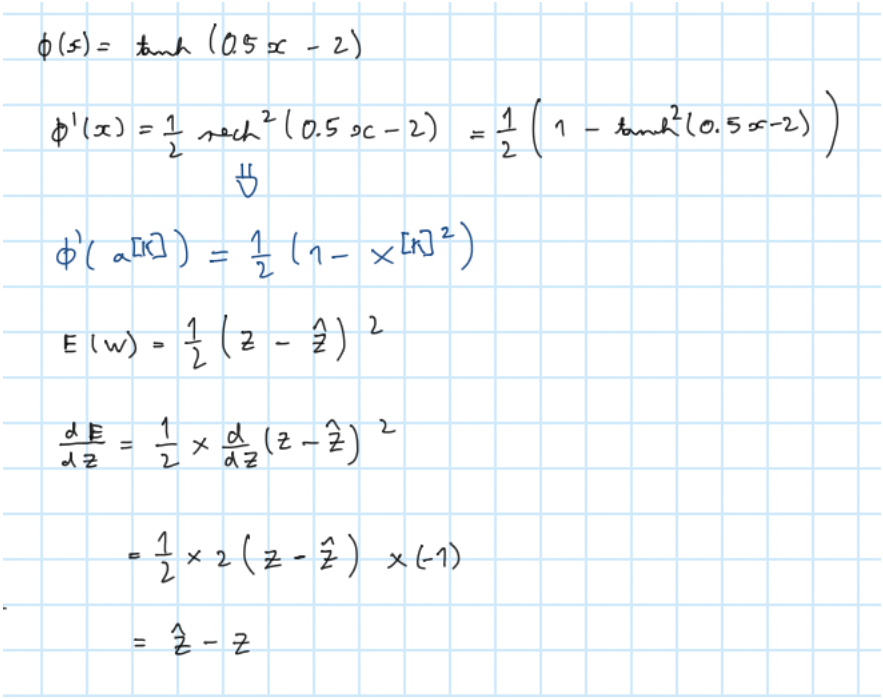
1. Forward Propagation

For starters, let's grab our input observations (x_1 and x_2) and propagate them across our MLP.

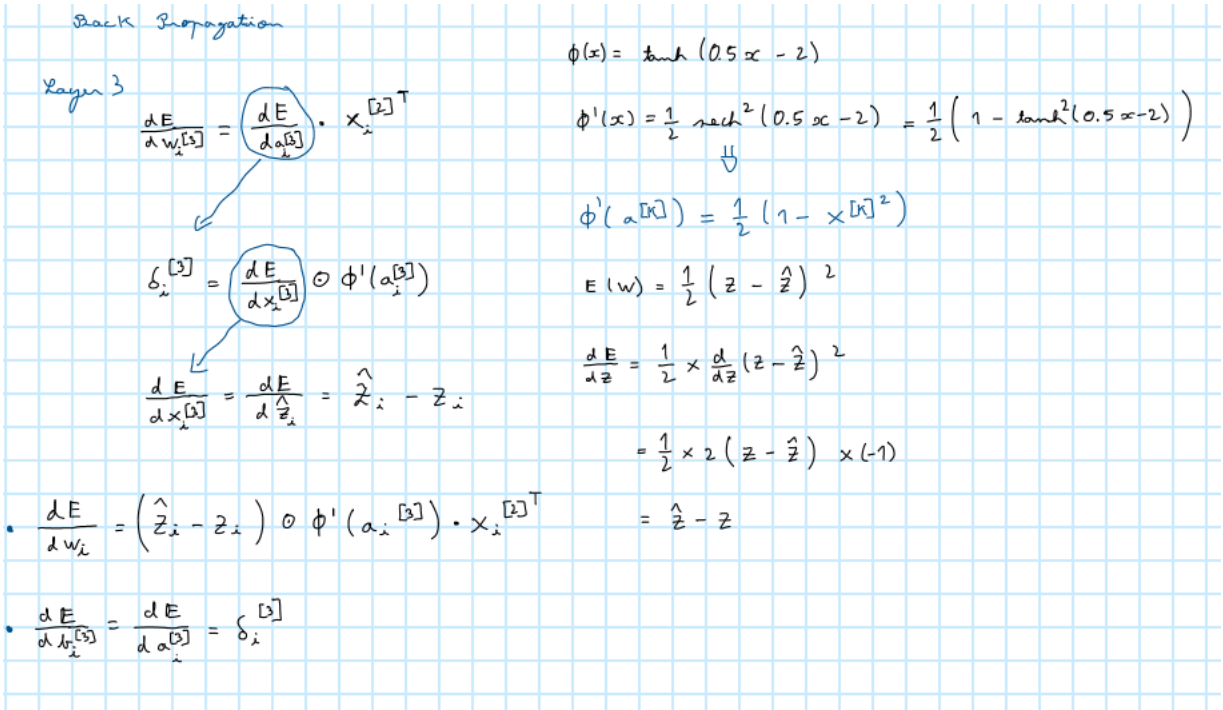


2. Back Propagation

Before we start this step, here are some auxiliary calculations/derivatives that will help us along the way.

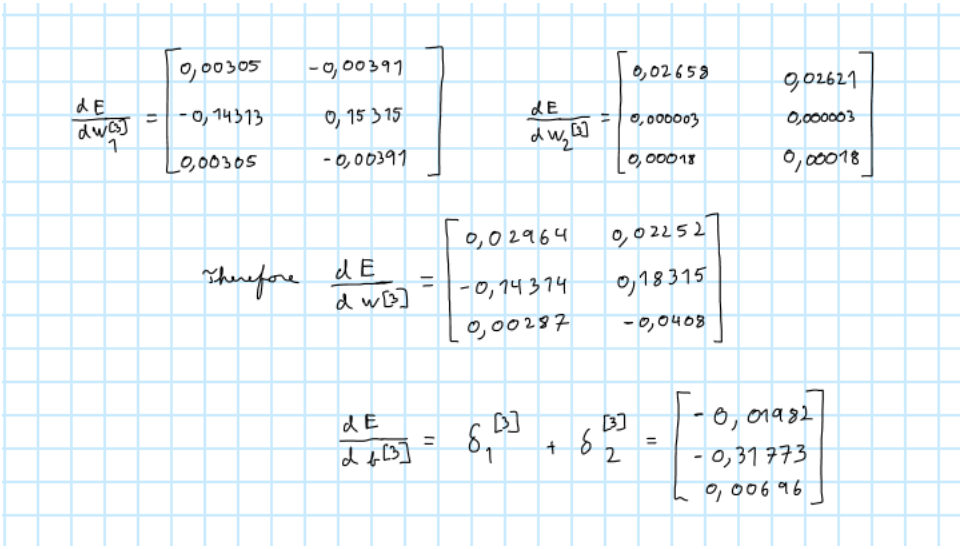


With that done, let's begin by finding the expressions required to calculate the gradient for the W[3] and the b[3] matrix.

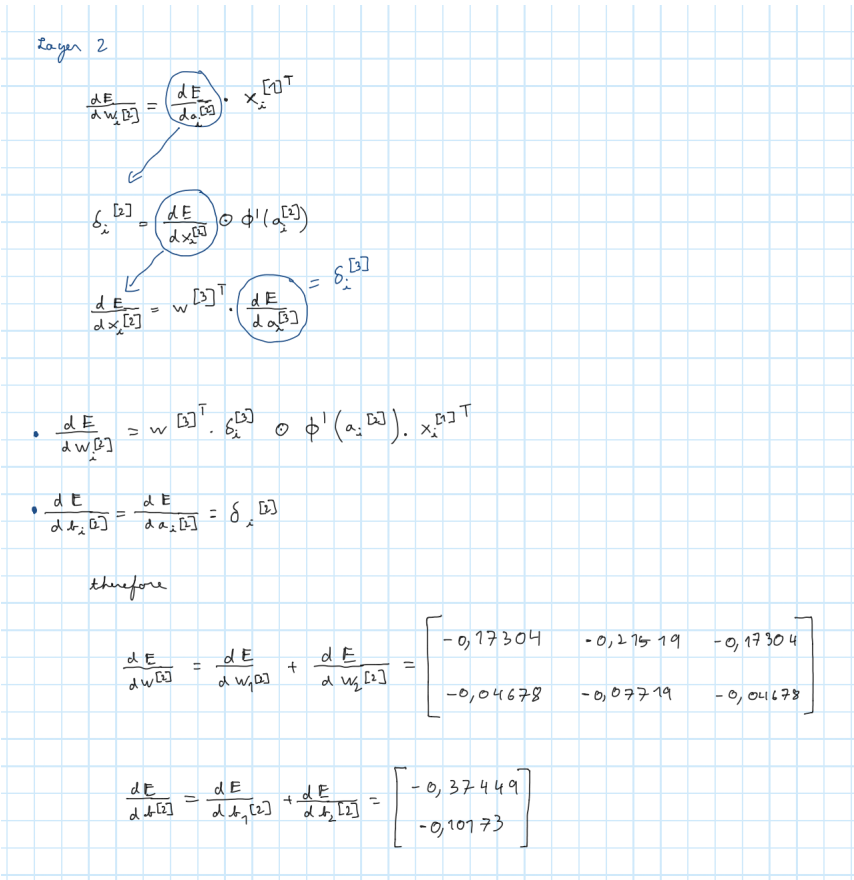


Since we are training with a single-batch gradient descent update, we will compute the gradients for

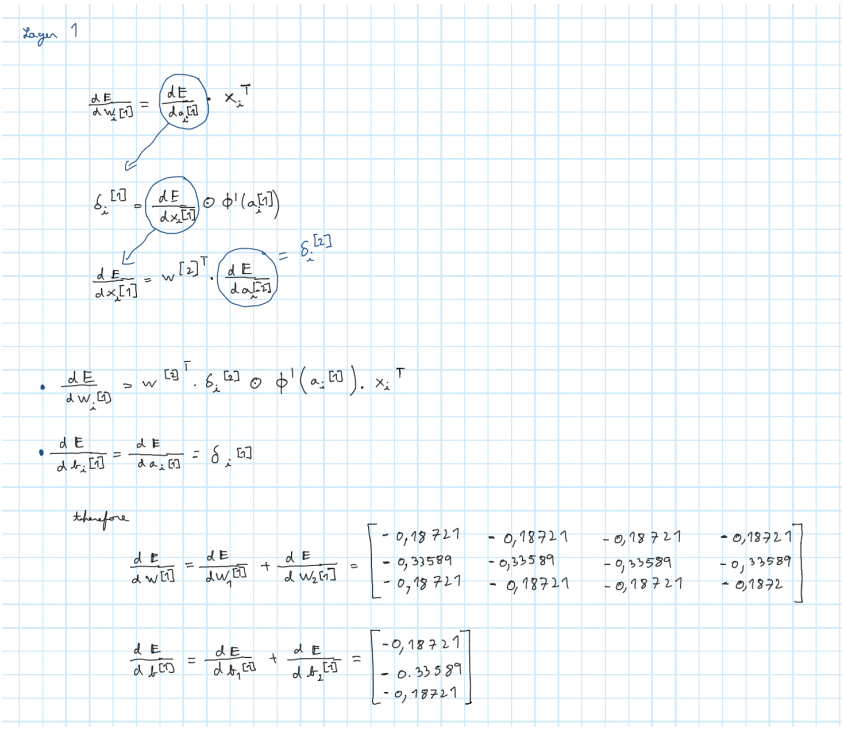
all our observations and then sum these gradients to determine the resulting adjustment. (Although not shown, we have also stored the delta values for future uses).



Afterwards, we proceed with layer 2, the logic being the same. Notice, how we know make use of the delta already calculated in the previous layer.

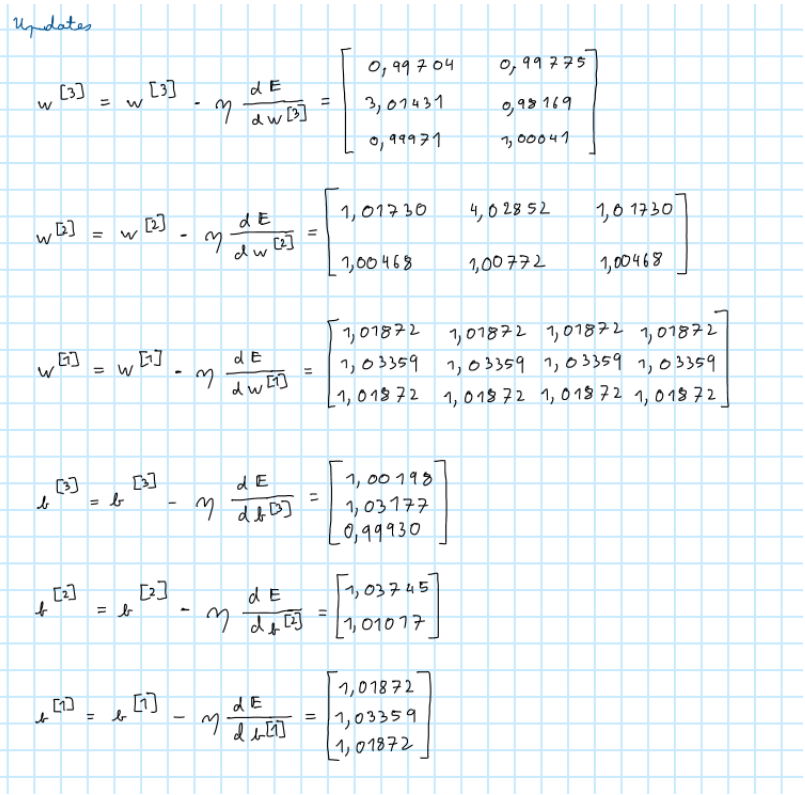


And finally, the adjustments for layer 1.



3. Updating the Weights/Biases

With all the previous steps done, we can obtain the new weights/biases by multiplying the computed adjustments with the learning rate and adding the result to the old value of the weights/biases.



Part II: Programming

In the next image, we have provided an overview of all the imports utilized and our approach to importing the dataset. Specifically, 'X' stores all of the input variables values, and 'y' stores the Output Class values (Wine quality). To streamline the code, we've omitted this information from the beginning of each section to avoid redundancy.

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import warnings

from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from math import sqrt

warnings.filterwarnings("ignore")

data = pd.read_csv('../data/winequality-red.csv', delimiter=';')

# Split the data into features (X) and target (y)
X = data.drop(columns=['quality'])
y = data['quality']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

1. Consider the *winequality − red.csv* dataset where the goal is to estimate the quality of a wine based on physicochemical inputs. Using a 80-20 training-test split with a fixed seed (random_state = 0), you are asked to learn MLP regressors to answer the following questions.

   (a) Learn a MLP regressor with 2 hidden layers of size 10, reLU activation and early stopping with 20% of training data set aside for validation. Plot the distribution of the residues (in absolute value) using a histogram.

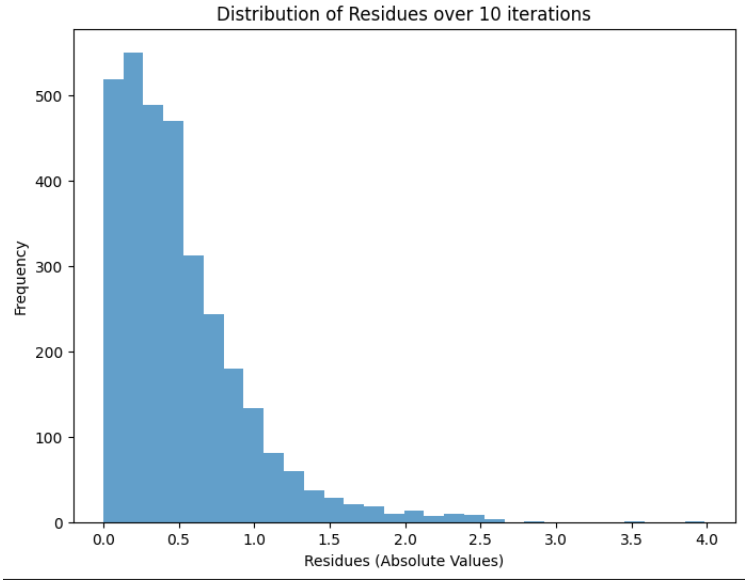
   Calculating the residuals:

```python
# Initialize lists to store residuals from each run
residuals = []

for run in range(1, 11):

    # Create and train the MLP regressor
    mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation='relu', validation_fraction=0.2, random_state=run, early_stopping=True)
    mlp.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = mlp.predict(X_test)

    # Calculate the absolute residuals
    residuals.append(np.abs(y_test - y_pred))
```

   Plotting them:

```python
# Plot a histogram of the residuals
plt.figure(figsize=(8, 6))
plt.hist(np.concatenate(residuals), bins=30, alpha=0.7)
plt.title('Distribution of Residues over 10 iterations')
plt.xlabel('Residues (Absolute Values)')
plt.ylabel('Frequency')
plt.show()
```

   Output:



(b) Assess the impact of rounding and bounding on the MAE of the MLP learnt in the previous question.

Calculating the Mean Absolute Error after rounding and bounding the predictions:

```
original_mae = []
rounded_bounded_mae = []

for run in range(1, 11):

    # Create and train the MLP regressor
    mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation="relu", validation_fraction=0.2, random_state=run, early_stopping=True)
    mlp.fit(X_train, y_train)

    # Make predictions on the test set
    y_pred = mlp.predict(X_test)

    # Round and bound the predictions
    y_pred_rounded_bounded = np.clip(np.round(y_pred), 1, 10)

    mae_original = mean_absolute_error(y_test, y_pred)
    mae_rounded_bounded = mean_absolute_error(y_test, y_pred_rounded_bounded)

    original_mae.append(mae_original)
    rounded_bounded_mae.append(mae_rounded_bounded)
```

Plotting:

```
# Calculate the average MAE for both original and rounded/bounded predictions
average_original_mae = np.mean(original_mae)
average_rounded_bounded_mae = np.mean(rounded_bounded_mae)

print("Average Original MAE:", average_original_mae)
print("Average Rounded/Bounded MAE:", average_rounded_bounded_mae)
```

Results:

MLP Model	**Average MAE**
Original MAE	0.50972
Rounded/Bounded MAE	0.43875

Assessment: The model using rounding/bounding is more accurate as we expected. This difference in performance can be attributed to the fact that bounding restricts predictions to values within the valid domain, while rounding enhances the model's accuracy by aligning predictions with the discrete nature of the target variable.

(c) Assess the impact on RMSE from replacing early stopping by a well-defined number of iterations in $\{20,50,100,200\}$.

Calculating the averaged RMSEs for each max iteration.

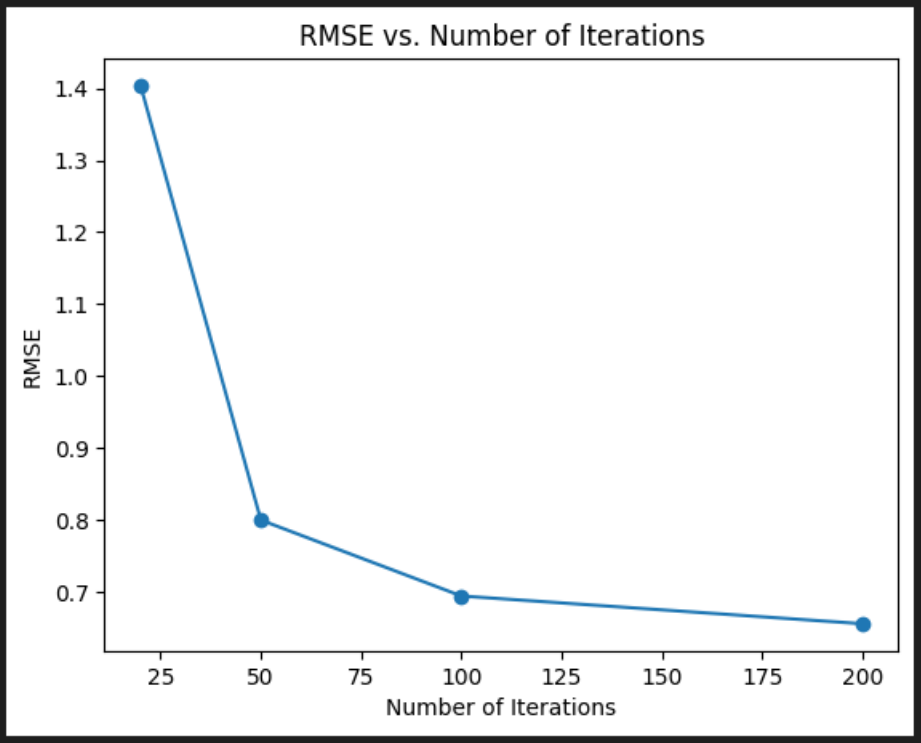
```
rmse_values = []

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
iteration_values = [20, 50, 100, 200]

for iteration in iteration_values:
    rmse_scores = []

    for run in range(1, 11):
        mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation='relu', max_iter=iteration, random_state=run)

        mlp.fit(X_train, y_train)

        prediction = mlp.predict(X_test)

        rmse = sqrt(mean_squared_error(y_test, prediction))
        rmse_scores.append(rmse)

    average_rmse = np.mean(rmse_scores)
    rmse_values.append(average_rmse)
```

Printing and plotting the averaged RMSEs for each iteration.

```
for i, num_iterations in enumerate(iteration_values):
    print(f"Average RMSE for {num_iterations} iterations: {rmse_values[i]:.2f}")

plt.plot(iteration_values, rmse_values, marker='o')
plt.title('RMSE vs. Number of Iterations')
plt.xlabel('Number of Iterations')
plt.ylabel('RMSE')
plt.show()
```

Output:

```
Average RMSE for 20 iterations: 1.40
Average RMSE for 50 iterations: 0.80
Average RMSE for 100 iterations: 0.69
Average RMSE for 200 iterations: 0.66
```



Comparison:

MLP Model	Averaged RMSE
Early Stopping	0.67
200 Iterations	0.66
100 Iterations	0.69
50 Iterations	0.80
20 Iterations	1.40

(d) Critically comment the results obtained in the previous question, hypothesizing at least one reason why early stopping favors and/or worsens performance.

Early stopping uses a small validation set to evaluate the model's accuracy. The moment the model's performance starts to deteriorate, early stopping terminates training. This way, the model generalizes better by not memorizing the entire dataset, helping reducing overfitting.

In question 3, we arrived at the same conclusion. When replacing early stopping with a fixed number of iterations, it's evident that the model's performance improves with the increase in the number of iterations. With too few iterations, the model tends to be underfitted, while with more iterations, it becomes more precise (until it starts overfitting).

We also calculated the average Root Mean Square Error (RMSE) for early stopping and compared it to the model with a fixed number of iterations. Interestingly, the early stopping approach resulted in a slightly larger average RMSE (0.67 vs. 0.66) when compared to the model with 200 iterations. This suggests that while early stopping generally enhances performance, there might be cases where the model achieves greater efficiency with a fixed number of iterations.