

Aprendizagem 2023
Homework III – Group 28

Gonalo Barias (ist1103124) & Raquel Braunschweig (ist1102624)

Part I: Pen and Paper

For questions in this group, show your numerical results with 5 decimals or scientific notation.

Hint: we highly recommend the use of `numpy` (e.g., `linalg.pinv` for inverse) or other programmatic facilities to support the calculus involved in both questions (1) and (2).

1. Consider the problem of learning a regression model from 4 bivariate observations

$$\left\{ \begin{pmatrix} 0.7 \\ -0.3 \end{pmatrix}, \begin{pmatrix} 0.4 \\ 0.5 \end{pmatrix}, \begin{pmatrix} -0.2 \\ 0.8 \end{pmatrix}, \begin{pmatrix} -0.4 \\ 0.3 \end{pmatrix} \right\} \text{ with targets } (0.8, 0.6, 0.3, 0.3).$$

- (a) **Given the radial basis function, $\phi_j(x) = \exp\left(-\frac{\|x - c_j\|^2}{2}\right)$ that transforms the original space onto a new space characterized by the similarity of the original observations to the following data points $\{c_1 = [0 \ 0]^T, c_2 = [1 \ -1]^T, c_3 = [-1 \ 1]^T\}$.**

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

Note: The intermediate values and matrices, in both items of this exercise, show values rounded to 5 decimal places, but all intermediate calculations have been carried out without rounding. The values were obtained with the help of `numpy` as suggested in the prompt. The code can be found in the Jupyter notebook `G028.ipynb`.

Considering the data points along with the 4 bivariate observations, we can calculate the values of $\phi_1(x)$, $\phi_2(x)$ and $\phi_3(x)$ for each observation, with $\phi_0(x)$ always being equal to 1.

x	$\phi_0(x)$	$\phi_1(x)$	$\phi_2(x)$	$\phi_3(x)$
$(0.7, -0.3)$	1	0.74826	0.74826	0.10127
$(0.4, 0.5)$	1	0.81465	0.27117	0.33121
$(-0.2, 0.8)$	1	0.71177	0.09633	0.71177
$(-0.4, 0.3)$	1	0.88250	0.16122	0.65377

Table 1: Value of $\phi_j(x)$ for each observation, $j = 0, \dots, 3$

Our goal is to perform a regression where the prediction is given by:

$$\hat{z}(x, w) = w_0 \phi_0(x) + w_1 \phi_1(x) + w_2 \phi_2(x) + w_3 \phi_3(x) \quad (1)$$

Since we want to learn a Ridge regression model, we will need to minimize the following regularized least-squares estimator:

$$E(w) = \frac{1}{2} \sum_{i=1}^N (z_i - w^T \cdot x_i)^2 + \frac{\lambda}{2} \|w\|_2^2 \quad (2)$$

To minimize (2), we set its gradient equal to 0 and obtain the closed-form solution with $\lambda = 0.1$ in equation (3). In the equation, we have that Φ is the matrix after applying $\phi_j(x)$ to the observations (Table 1) and z is the vector of targets for the observations, $z = [0.8 \ 0.6 \ 0.3 \ 0.3]^T$.

$$\nabla E(w) = 0 \Leftrightarrow w = (\Phi^T \Phi + \lambda I)^{-1} \cdot \Phi^T z \Leftrightarrow w = (\Phi^T \Phi + 0.1I)^{-1} \cdot \Phi^T z \quad (3)$$

$$\Phi = \begin{bmatrix} 1.00000 & 0.74826 & 0.74826 & 0.10127 \\ 1.00000 & 0.81465 & 0.27117 & 0.33121 \\ 1.00000 & 0.71177 & 0.09633 & 0.71177 \\ 1.00000 & 0.88250 & 0.16122 & 0.65377 \end{bmatrix} \quad \Phi^T = \begin{bmatrix} 1.00000 & 1.00000 & 1.00000 & 1.00000 \\ 0.74826 & 0.81465 & 0.71177 & 0.88250 \\ 0.74826 & 0.27117 & 0.09633 & 0.16122 \\ 0.10127 & 0.33121 & 0.71177 & 0.65377 \end{bmatrix}$$

Therefore, we can then calculate the remaining values, until we get the vector w :

$$\begin{aligned} \Phi^T \Phi &= \begin{bmatrix} 4.00000 & 3.15718 & 1.27698 & 1.79802 \\ 3.15718 & 2.50897 & 0.99165 & 1.42916 \\ 1.27698 & 0.99165 & 0.66870 & 0.33955 \\ 1.79802 & 1.42917 & 0.33956 & 1.05399 \end{bmatrix} \\ \Phi^T \Phi + 0.1 I &= \begin{bmatrix} 4.10000 & 3.15718 & 1.27698 & 1.79802 \\ 3.15718 & 2.60897 & 0.99165 & 1.42917 \\ 1.27698 & 0.99165 & 0.76870 & 0.33956 \\ 1.79802 & 1.42917 & 0.33956 & 1.15399 \end{bmatrix} \\ (\Phi^T \Phi + 0.1I)^{-1} &= \begin{bmatrix} 4.54826 & -3.77682 & -1.86117 & -1.86155 \\ -3.77682 & 5.98285 & -0.88543 & -1.26432 \\ -1.86117 & -0.88543 & 4.33276 & 2.72156 \\ -1.86155 & -1.26432 & 2.72156 & 4.53204 \end{bmatrix} \\ (\Phi^T \Phi + 0.1I)^{-1} \cdot \Phi^T &= \begin{bmatrix} 0.14105 & 0.35022 & 0.35575 & -0.30185 \\ -0.09064 & 0.43823 & -0.50361 & 0.53370 \\ 0.99394 & -0.50615 & -0.13690 & -0.16477 \\ -0.31222 & -0.65246 & 0.72647 & 0.42436 \end{bmatrix} \\ w = (\Phi^T \Phi + 0.1I)^{-1} \cdot \Phi^T z &= \begin{bmatrix} 0.33914 & 0.19945 & 0.40096 & -0.29600 \end{bmatrix}^T \end{aligned}$$

Having calculated the vector w , we now know the values of each weight in the regression:

$$w_0 = 0.33914 \quad w_1 = 0.19945 \quad w_2 = 0.40096 \quad w_3 = -0.29600$$

Finally, replacing them in (1) gives us the regression expression:

$$\hat{z}(x, w) = 0.33914 + 0.19945 \phi_1(x) + 0.40096 \phi_2(x) - 0.29600 \phi_3(x) \quad (4)$$

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

The RMSE (Root Mean Square Error) is given by:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (z_i - \hat{z}_i)^2} \quad (5)$$

Here we have $N = 4$, because 4 is the number of samples in the dataset, z_i is the true label for the i -th sample and \hat{z}_i is the predicted label for the i -th sample.

Using (4) from the previous item, we can determine \hat{z} for the 4 observations:

$$\Phi = \begin{bmatrix} 1.00000 & 0.74826 & 0.74826 & 0.10127 \\ 1.00000 & 0.81465 & 0.27117 & 0.33121 \\ 1.00000 & 0.71177 & 0.09633 & 0.71177 \\ 1.00000 & 0.88250 & 0.16122 & 0.65377 \end{bmatrix} \wedge w = \begin{bmatrix} 0.33914 \\ 0.19945 \\ 0.40096 \\ -0.29600 \end{bmatrix} \Rightarrow \hat{z} = \Phi w = \begin{bmatrix} 0.75844 \\ 0.51232 \\ 0.30905 \\ 0.38629 \end{bmatrix}$$

Finally, we can take $z = [0.8 \ 0.6 \ 0.3 \ 0.3]^T$ along with \hat{z} and calculate RMSE, using (5):

$$\text{RMSE} = \sqrt{\frac{1}{4} \times \sum_{i=1}^4 (z_i - \hat{z}_i)^2} = \sqrt{\frac{1}{4} \times 0.01694} = \mathbf{0.06508}$$

2. **Consider a MLP classifier of three outcomes - A, B and C - characterized by the following weights and activation function for every unit:**

$$W^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, b^{[1]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, W^{[2]} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix}, b^{[2]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, W^{[3]} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \\ 1 & 1 \end{bmatrix}, b^{[3]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$f(x) = \frac{e^{0.5x-2} - e^{-0.5x+2}}{e^{0.5x-2} + e^{-0.5x+2}} = \tanh(0.5x - 2)$$

We also have the squared error loss $\frac{1}{2}\|z - \hat{z}\|_2^2$. Perform one batch gradient descent update (with learning rate $\eta = 0.1$) for training observations $x_1 = [1 \ 1 \ 1 \ 1]^T$ and $x_2 = [1 \ 0 \ 0 \ -1]^T$ with targets B and A, respectively.

Note: To help with the calculations and as suggested in the prompt, we used numpy. The code can be found in the Jupyter notebook `G028.ipynb`.

To begin, we initiate the process with the **forward propagation**. Below are the key equations to calculate the values of each layer, $x_i^{[p]}$, per observation:

$$z_i^{[p]} = W^{[p]} x_i^{[p-1]} + b^{[p]} \quad x_i^{[p]} = f(z_i^{[p]})$$

The function f is the activation function, for every layer, provided in the prompt, $f(z_i^{[p]}) = \tanh(0.5 z_i^{[p]} - 2)$. In this notation, the value p is the index of the MLP layer and n is the number of the observation, either 1 or 2.

We can now calculate the values of each node in the multi-layer perceptron with the initialized weights and biases provided in the prompt, for each of the observations.

We will **start** with x_1 :

$$z_1^{[1]} = W^{[1]} x_1^{[0]} + b^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \\ 5 \end{bmatrix}$$

$$\begin{aligned}
x_1^{[1]} &= f(z_1^{[1]}) = \tanh(0.5 z_1^{[1]} - 2) \approx \begin{bmatrix} 0.46212 \\ 0.76159 \\ 0.46212 \end{bmatrix} \\
z_1^{[2]} &= W^{[2]} x_1^{[1]} + b^{[2]} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.46212 \\ 0.76159 \\ 0.46212 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 4.97061 \\ 2.68583 \end{bmatrix} \\
x_1^{[2]} &= f(z_1^{[2]}) = \tanh(0.5 z_1^{[2]} - 2) \approx \begin{bmatrix} 0.45048 \\ -0.57642 \end{bmatrix} \\
z_1^{[3]} &= W^{[3]} x_1^{[2]} + b^{[3]} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0.45048 \\ -0.57642 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 0.87406 \\ 1.77503 \\ 0.87406 \end{bmatrix} \\
x_1^{[3]} &= f(z_1^{[3]}) = \tanh(0.5 z_1^{[3]} - 2) \approx \begin{bmatrix} -0.9159 \\ -0.80494 \\ -0.9159 \end{bmatrix}
\end{aligned}$$

Next in line is x_2 :

$$\begin{aligned}
z_2^{[1]} &= W^{[1]} x_2^{[0]} + b^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\
x_2^{[1]} &= f(z_2^{[1]}) = \tanh(0.5 z_2^{[1]} - 2) \approx \begin{bmatrix} -0.90515 \\ -0.90515 \\ -0.90515 \end{bmatrix} \\
z_2^{[2]} &= W^{[2]} x_2^{[1]} + b^{[2]} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -0.90515 \\ -0.90515 \\ -0.90515 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} -4.43089 \\ -1.71544 \end{bmatrix} \\
x_2^{[2]} &= f(z_2^{[2]}) = \tanh(0.5 z_2^{[2]} - 2) \approx \begin{bmatrix} -0.99956 \\ -0.99343 \end{bmatrix} \\
z_2^{[3]} &= W^{[3]} x_2^{[2]} + b^{[3]} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -0.99956 \\ -0.99343 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} -0.993 \\ -2.99212 \\ -0.993 \end{bmatrix} \\
x_2^{[3]} &= f(z_2^{[3]}) = \tanh(0.5 z_2^{[3]} - 2) \approx \begin{bmatrix} -0.98652 \\ -0.99816 \\ -0.98652 \end{bmatrix}
\end{aligned}$$

Now that we have obtained the values for our observations, considering the initial weights and biases, we can initiate the **back propagation** process to perform a batch gradient descent update (with learning rate $\eta = 0.1$).

The updated values for the weights and biases can be obtained by the following equations:

$$W_{new}^{[i]} = W_{old}^{[i]} + \Delta W^{[i]} = W_{old}^{[i]} - \eta \frac{\partial E}{\partial W^{[i]}} = W_{old}^{[i]} - 0.1 \frac{\partial E}{\partial W^{[i]}} \quad (6)$$

$$b_{new}^{[i]} = b_{old}^{[i]} + \Delta b^{[i]} = b_{old}^{[i]} - \eta \frac{\partial E}{\partial b^{[i]}} = b_{old}^{[i]} - 0.1 \frac{\partial E}{\partial b^{[i]}} \quad (7)$$

We first need to take into account the error function, which, as per the prompt, is the half squared error loss, defined as follows:

$$E(w) = \frac{1}{2} \|\mathbf{z} - \hat{\mathbf{z}}\|_2^2 = \frac{1}{2} \sum_{i=1}^N (t_i - o_i)^2 \quad (8)$$

The next step involves computing its derivatives by applying the **chain rule**:

$$\frac{\partial E}{\partial W^{[p]}} = \sum_{i=1}^2 \left(\frac{\partial E}{\partial x_i^{[p]}} \circ \frac{\partial x_i^{[p]}}{\partial z_i^{[p]}} \cdot \frac{\partial z_i^{[p]}}{\partial W^{[p]}} \right) \quad \frac{\partial E}{\partial b^{[p]}} = \sum_{i=1}^2 \left(\frac{\partial E}{\partial x_i^{[p]}} \circ \frac{\partial x_i^{[p]}}{\partial z_i^{[p]}} \cdot \frac{\partial z_i^{[p]}}{\partial b^{[p]}} \right) \quad (9)$$

This can be broken down into:

$$\begin{aligned} \frac{\partial E}{\partial x_i^{[p]}} \circ \frac{\partial x_i^{[p]}}{\partial z_i^{[p]}} &= \delta_i^{[p]} \\ \frac{\partial E}{\partial x_i^{[3]}} &= \frac{1}{2} \times 2 \times (x_i^{[3]} - t_i) = x_i^{[3]} - t_i \\ \frac{\partial E}{\partial x_i^{[p]}} &= \left(W^{[p+1]} \right)^T \cdot \delta_i^{[p+1]}, \quad p \neq 3 \quad (\text{using recursion}) \\ \frac{\partial x_i^{[p]}}{\partial z_i^{[p]}} &= f' \left(z_i^{[p]} \right) = \tanh' \left(0.5 \times z_i^{[p]} - 2 \right) \times \frac{d}{dz_i^{[p]}} \left(0.5 \times z_i^{[p]} - 2 \right) \\ &= \text{sech}^2 \left(0.5 \times z_i^{[p]} - 2 \right) \times 0.5 \\ \frac{\partial z_i^{[p]}}{\partial W^{[p]}} &= \left(x_i^{[p-1]} \right)^T \quad \frac{\partial z_i^{[p]}}{\partial b^{[p]}} = 1 \end{aligned}$$

Therefore, the derivatives of the error function are given by:

$$\frac{\partial E}{\partial W^{[p]}} = \sum_{i=1}^2 \left(\delta_i^{[p]} \cdot \left(x_i^{[p-1]} \right)^T \right) \quad \frac{\partial E}{\partial b^{[p]}} = \sum_{i=1}^2 \left(\delta_i^{[p]} \right) \quad (10)$$

When computing δ , we split it into two separate cases: if it is in last layer (11), or in non-external layers (12):

$$\delta_i^{[3]} = \left(x_i^{[3]} - t_i \right) \circ \text{sech}^2 \left(0.5 \times z_i^{[3]} - 2 \right) \times 0.5 \quad (11)$$

$$\delta_i^{[p]} = \left(\left(W^{[p+1]} \right)^T \cdot \delta_i^{[p+1]} \right) \circ \text{sech}^2 \left(0.5 \times z_i^{[p]} - 2 \right) \times 0.5, \quad p \neq 3 \quad (12)$$

Since, the \tanh function has a codomain in $[-1, 1]$ and since the targets for x_1 and x_2 are B and A, respectively, we can conclude the following encodings for the outputs:

$$t_1 = \begin{bmatrix} -1 & 1 & -1 \end{bmatrix}^T \quad t_2 = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix}^T$$

Now, we have all we need to compute the values for the deltas, δ , by replacing the equation on (11) for $p = 3$, and (12) for any $p \neq 3$:

$$\begin{aligned} \delta_1^{[1]} &= \left(\left(W^{[2]} \right)^T \cdot \delta_1^{[2]} \right) \circ \text{sech}^2 \left(0.5 \times z_1^{[1]} - 2 \right) \times 0.5 = \\ &= \left(\begin{bmatrix} 141 \\ 111 \end{bmatrix}^T \cdot \begin{bmatrix} -0.37448 \\ -0.10156 \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} 5 \\ 6 \\ 5 \end{bmatrix} - 2 \right) \times 0.5 = \\ &= \begin{bmatrix} -0.18719 \\ -0.33587 \\ -0.18719 \end{bmatrix} \\ \delta_2^{[1]} &= \left(\left(W^{[2]} \right)^T \cdot \delta_2^{[2]} \right) \circ \text{sech}^2 \left(0.5 \times z_2^{[1]} - 2 \right) \times 0.5 = \\ &= \left(\begin{bmatrix} 141 \\ 111 \end{bmatrix}^T \cdot \begin{bmatrix} -1 \times 10^{-5} \\ -1.7 \times 10^{-4} \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - 2 \right) \times 0.5 = \\ &= \begin{bmatrix} -2 \times 10^{-5} \\ -2 \times 10^{-5} \\ -2 \times 10^{-5} \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \delta_1^{[2]} &= \left(\left(W^{[3]} \right)^T \cdot \delta_1^{[3]} \right) \circ \text{sech}^2 \left(0.5 \times z_1^{[2]} - 2 \right) \times 0.5 = \\ &= \left(\begin{bmatrix} 11 \\ 31 \\ 11 \end{bmatrix}^T \cdot \begin{bmatrix} 0.00678 \\ -0.31773 \\ 0.00678 \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} 4.97061 \\ 2.68583 \end{bmatrix} - 2 \right) \times 0.5 = \\ &= \begin{bmatrix} -0.37448 \\ -0.10156 \end{bmatrix} \\ \delta_2^{[2]} &= \left(\left(W^{[3]} \right)^T \cdot \delta_2^{[3]} \right) \circ \text{sech}^2 \left(0.5 \times z_2^{[2]} - 2 \right) \times 0.5 = \\ &= \left(\begin{bmatrix} 11 \\ 31 \\ 11 \end{bmatrix}^T \cdot \begin{bmatrix} -0.0266 \\ 0 \\ -0.00018 \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} 4.97061 \\ 2.68583 \end{bmatrix} - 2 \right) \times 0.5 = \\ &= \begin{bmatrix} -1 \times 10^{-5} \\ -1.7 \times 10^{-4} \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \delta_1^{[3]} &= \left(x_1^{[3]} - t_1 \right) \circ \text{sech}^2 \left(0.5 \times z_1^{[3]} - 2 \right) \times 0.5 = \\ &= \left(\begin{bmatrix} -0.9159 \\ -0.80494 \\ -0.9159 \end{bmatrix} - \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} 0.87406 \\ 1.77503 \\ 0.87406 \end{bmatrix} \times 0.5 \right) = \end{aligned}$$

$$\begin{aligned}
&= \begin{bmatrix} 0.00678 \\ -0.31773 \\ 0.00678 \end{bmatrix} \\
\delta_2^{[3]} &= (x_2^{[3]} - t_2) \circ \text{sech}^2(0.5 \times z_2^{[3]} - 2) \times 0.5 = \\
&= \left(\begin{bmatrix} -0.98652 \\ -0.99816 \\ -0.98652 \end{bmatrix} - \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \right) \circ \text{sech}^2 \left(0.5 \times \begin{bmatrix} -0.993 \\ -2.99212 \\ -0.993 \end{bmatrix} \times 0.5 \right) = \\
&= \begin{bmatrix} -0.0266 \\ 0 \\ 0.00018 \end{bmatrix}
\end{aligned}$$

Next, we will compute the derivatives of the error function, employing the first equation at (10):

$$\begin{aligned}
\frac{\partial E}{\partial W^{[1]}} &= \sum_{i=1}^2 \left(\delta_i^{[1]} \cdot (x_i^{[0]})^T \right) = \left(\delta_1^{[1]} \cdot (x_1^{[0]})^T \right) + \left(\delta_2^{[1]} \cdot (x_2^{[0]})^T \right) = \\
&= \left(\begin{bmatrix} -0.18719 \\ -0.33587 \\ -0.18719 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}^T \right) + \left(\begin{bmatrix} -2 \times 10^{-5} \\ -2 \times 10^{-5} \\ -2 \times 10^{-5} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}^T \right) = \\
&= \begin{bmatrix} -0.18721 & -0.18719 & -0.18719 & -0.18717 \\ -0.33589 & -0.33587 & -0.33587 & -0.33585 \\ -0.18721 & -0.18719 & -0.18719 & -0.18717 \end{bmatrix} \\
\frac{\partial E}{\partial W^{[2]}} &= \sum_{i=1}^2 \left(\delta_i^{[2]} \cdot (x_i^{[1]})^T \right) = \left(\delta_1^{[2]} \cdot (x_1^{[1]})^T \right) + \left(\delta_2^{[2]} \cdot (x_2^{[1]})^T \right) = \\
&= \left(\begin{bmatrix} -0.37448 \\ 0.10156 \end{bmatrix} \cdot \begin{bmatrix} 0.46212 \\ 0.76159 \\ 0.46212 \end{bmatrix}^T \right) + \left(\begin{bmatrix} -1 \times 10^{-5} \\ -1.7 \times 10^{-4} \end{bmatrix} \cdot \begin{bmatrix} -0.90515 \\ -0.90515 \\ -0.90515 \end{bmatrix}^T \right) = \\
&= \begin{bmatrix} -0.17304 & -0.28519 & -0.17304 \\ -0.04678 & -0.07719 & -0.04678 \end{bmatrix} \\
\frac{\partial E}{\partial W^{[3]}} &= \sum_{i=1}^2 \left(\delta_i^{[3]} \cdot (x_i^{[2]})^T \right) = \left(\delta_1^{[3]} \cdot (x_1^{[2]})^T \right) + \left(\delta_2^{[3]} \cdot (x_2^{[2]})^T \right) = \\
&= \left(\begin{bmatrix} 0.00678 \\ -0.31773 \\ 0.00678 \end{bmatrix} \cdot \begin{bmatrix} 0.45048 \\ -0.57642 \end{bmatrix}^T \right) + \left(\begin{bmatrix} -0.0266 \\ 0 \\ 0.00018 \end{bmatrix} \cdot \begin{bmatrix} -0.99956 \\ -0.99343 \end{bmatrix}^T \right) = \\
&= \begin{bmatrix} 0.02964 & 0.02252 \\ -0.14314 & 0.18315 \\ 0.00287 & -0.00408 \end{bmatrix}
\end{aligned}$$

Next, we will employ the second equation at (10):

$$\frac{\partial E}{\partial b^{[1]}} = \sum_{i=1}^2 \left(\delta_i^{[1]} \right) = \delta_1^{[1]} + \delta_2^{[1]} = \begin{bmatrix} -0.18719 \\ -0.33587 \\ -0.18719 \end{bmatrix} + \begin{bmatrix} -2 \times 10^{-5} \\ -2 \times 10^{-5} \\ -2 \times 10^{-5} \end{bmatrix} = \begin{bmatrix} -0.18721 \\ -0.33589 \\ -0.18721 \end{bmatrix}$$

$$\frac{\partial E}{\partial b^{[2]}} = \sum_{i=1}^2 \left(\delta_i^{[2]} \right) = \delta_1^{[2]} + \delta_2^{[2]} = \begin{bmatrix} -0.37448 \\ 0.10156 \end{bmatrix} + \begin{bmatrix} -1 \times 10^{-5} \\ -1.7 \times 10^{-4} \end{bmatrix} = \begin{bmatrix} -0.37449 \\ -0.10173 \end{bmatrix}$$

$$\frac{\partial E}{\partial b^{[3]}} = \sum_{i=1}^2 \left(\delta_i^{[3]} \right) = \delta_1^{[3]} + \delta_2^{[3]} = \begin{bmatrix} 0.00678 \\ -0.31773 \\ 0.00678 \end{bmatrix} + \begin{bmatrix} -0.0266 \\ 0 \\ 0.00018 \end{bmatrix} = \begin{bmatrix} -0.01982 \\ -0.31773 \\ 0.00696 \end{bmatrix}$$

The final step is to calculate the updated weights and biases.

By substituting the values into (6), we obtain the following updated weights:

$$W_{new}^{[1]} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.18721 & -0.18719 & -0.18719 & -0.18717 \\ -0.33589 & -0.33587 & -0.33587 & -0.33585 \\ -0.18721 & -0.18719 & -0.18719 & -0.18717 \end{bmatrix} =$$

$$= \begin{bmatrix} -1.01872 & -1.01872 & -1.01872 & -1.01872 \\ 1.03359 & 1.03359 & 2.03359 & 1.03359 \\ 1.01872 & 1.01872 & 1.01872 & 1.01872 \end{bmatrix}$$

$$W_{new}^{[2]} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 1 & 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.17304 & -0.28519 & -0.17304 \\ -0.04678 & -0.07719 & -0.04678 \end{bmatrix} =$$

$$= \begin{bmatrix} 1.0173 & 4.02852 & 1.0173 \\ 1.00468 & 1.00772 & 1.00468 \end{bmatrix}$$

$$W_{new}^{[3]} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \\ 1 & 1 \end{bmatrix} - 0.1 \begin{bmatrix} 0.02964 & 0.02252 \\ -0.14314 & 0.18315 \\ 0.00287 & -0.00408 \end{bmatrix} =$$

$$= \begin{bmatrix} 0.99704 & 0.99775 \\ 3.01431 & 0.98169 \\ 0.99971 & 1.00041 \end{bmatrix}$$

Finally, by replacing the values into (7), we get the following updated biases:

$$b_{new}^{[1]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.18721 \\ -0.33589 \\ -0.18721 \end{bmatrix} = \begin{bmatrix} 1.01872 \\ 1.03359 \\ 1.01872 \end{bmatrix}$$

$$b_{new}^{[2]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.37449 \\ -0.10173 \end{bmatrix} = \begin{bmatrix} 1.03745 \\ 1.01017 \end{bmatrix}$$

$$b_{new}^{[3]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.01982 \\ -0.31773 \\ 0.00696 \end{bmatrix} = \begin{bmatrix} 1.00198 \\ 1.03177 \\ 0.9993 \end{bmatrix}$$

Part II: Programming and critical analysis

Considering the `winequality-red.csv` dataset (available at the webpage) where the goal is to estimate the quality (sensory appreciation) 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.

Given their stochastic behavior, average the performance of each MLP from 10 runs (for reproducibility consider seeding the MLPs with `random_state ∈ {1..10}`).

1. **Learn a MLP regressor with 2 hidden layers of size 10, rectifier linear unit activation on all nodes, and early stopping with 20% of training data set aside for validation. All remaining parameters (e.g., loss, batch size, regularization term, solver) should be set as default. Plot the distribution of the residues (in absolute value) using a histogram.**

```
1 import numpy as np, pandas as pd, matplotlib.pyplot as plt
2 from sklearn.model_selection import train_test_split
3 from sklearn.neural_network import MLPRegressor
4
5 # Step 1: Load and prepare the dataset
6 data = pd.read_csv("./data/winequality-red.csv", sep=";")
7 X, y = data.drop("quality", axis=1), data["quality"]
8 X_train, X_test, y_train, y_test = train_test_split(X, y,
9                                                    test_size=0.2, random_state=0)
10
11 residues = []
12 for rs in range(1, 11):
13     # Step 2: Learn the MLP regressor
14     mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation="relu",
15                        early_stopping=True, validation_fraction=0.2,
16                        random_state=rs)
17     mlp.fit(X_train, y_train)
18
19     # Step 3: Collect the residues
20     y_pred = mlp.predict(X_test)
21     residues.extend(np.abs(y_pred - y_test))
22
23 # Step 4: Plot the distribution of the absolute residues
24 plt.figure(figsize=(8, 6))
25 plt.hist(residues, bins=30, color = "#00bfc4", edgecolor="black")
26 plt.xlabel("Absolute Residues")
27 plt.ylabel("Count")
28 plt.show()
```

The residues were calculated as the absolute difference between the predicted value and the expected value, for each observation. Due to the stochastic behavior of MLP regressors, we took the results of the MLP from 10 runs.

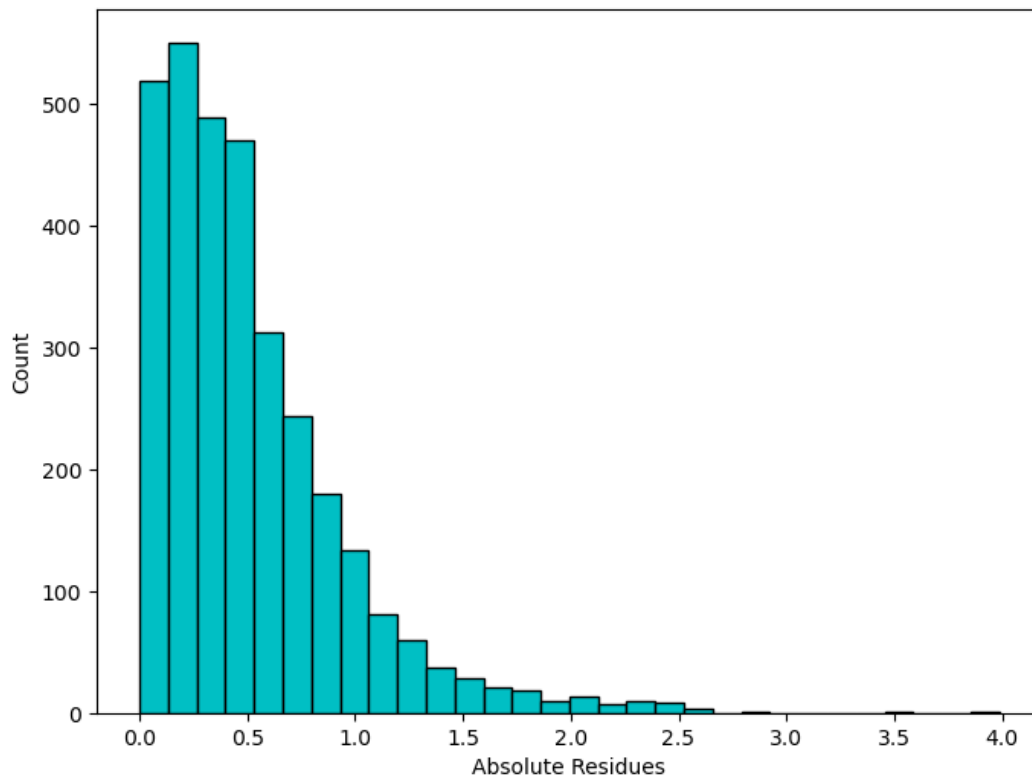


Figure 1: The distribution of absolute residues for MLP regression on wine quality prediction using an histogram with 30 bins

2. Since we are in the presence of a *integer regression* task, a recommended trick is to round and bound estimates. Assess the impact of these operations on the MAE of the MLP learnt in the previous question.

```

1 import numpy as np, pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.neural_network import MLPRegressor
4 from sklearn.metrics import mean_absolute_error
5
6 # Just like in the previous exercise
7 data = pd.read_csv("./data/winequality-red.csv", sep=";")
8 X, y = data.drop("quality", axis=1), data["quality"]
9 X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                                    test_size=0.2, random_state=0)
11
12 maes, maes_rb = [], []
13 for rs in range(1, 11):
14     # Train the MLP regressor with a specific number of iterations
15     mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation="relu",
16                        early_stopping=True, validation_fraction=0.2,
17                        random_state=rs)
18     mlp.fit(X_train, y_train)
19
20     # Apply rounding and bounding operations
21     y_pred = mlp.predict(X_test)
22     y_pred_rb = np.clip(np.round(y_pred), 1, 10) # Bound between 1 and 10

```

```

23
24 # Calculate MAE for both rounded and bounded predictions
25 maes.append(mean_absolute_error(y_test, y_pred))
26 maes_rb.append(mean_absolute_error(y_test, y_pred_rb))
27
28 # Print the MAE for both cases
29 print(f"MAE without operations: {np.mean(maes)}")
30 print(f"MAE with rounded and bounded predictions: {np.mean(maes_rb)}")

```

MAE without operations: 0.5097171955009515

MAE with rounded and bounded predictions: 0.43875

By rounding to the nearest unit and bounding the estimates between 1 and 10 (as per the *FAQ*), we get a lower Mean Absolute Error. The wine quality is an integer between 1 and 10, so it is expected that rounding and bounding the estimates gets them closer to the real integer values.

3. Similarly assess the impact on RMSE from replacing early stopping by a well-defined number of iterations in {20, 50, 100, 200} (where one iteration corresponds to a batch).

```

1 import numpy as np, pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.neural_network import MLPRegressor
4 from sklearn.metrics import mean_squared_error
5
6 # Just like in the previous exercise
7 data = pd.read_csv("./data/winequality-red.csv", sep=";")
8 X, y = data.drop("quality", axis=1), data["quality"]
9 X_train, X_test, y_train, y_test = train_test_split(X, y,
10                                                    test_size=0.2, random_state=0)
11
12 n_iters, rmse_iters = [20, 50, 100, 200], []
13 for n_iter in n_iters:
14     rmse_runs = []
15     for rs in range(1, 11):
16         # Train the MLP regressor with a specific number of iterations
17         mlp = MLPRegressor(hidden_layer_sizes=(10, 10), activation="relu",
18                             max_iter=n_iter, validation_fraction=0.2,
19                             random_state=rs)
20         mlp.fit(X_train, y_train)
21
22         # Predict the target values on the test set
23         y_pred = mlp.predict(X_test)
24
25         # Calculate RMSE
26         rmse_runs.append(mean_squared_error(y_test, y_pred, squared=False))
27     rmse_iters.append(np.mean(rmse_runs))
28
29 # Print the RMSE for the different numbers of iterations
30 for i, n_iter in enumerate(n_iters):
31     print(f"RMSE with {n_iter} iterations: {rmse_iters[i]}")

```

RMSE with 20 iterations: 1.4039789509925442

RMSE with 50 iterations: 0.7996073631460567

RMSE with 100 iterations: 0.6940361469112144

RMSE with 200 iterations: 0.6554543932216472

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

Increasing the number of training cycles consistently lowered the Root Mean Squared Error (RMSE). This is expected because more cycles let the model adjust its weights and bias, potentially fitting the data accurately.

Now, let's understand how stopping training early might affect performance:

- **Possible downside:** If we stop too soon, we might halt before the model reaches its optimal performance.
- **Potential benefit:** Early stopping can prevent the model from overfitting, which can help it make better predictions on new and unseen data.