Aprendizagem 2023
Homework I — Group 003
(ist1107028, ist1107137)

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).

Below is a training dataset D composed by two input variables and two output variables, one of which is numerical ($\mathbf{y_{num}}$) and the other categorical ($\mathbf{y_{class}}$) . Consider a polynomial basis function $\phi(\mathbf{y_1}, \mathbf{y_2}) = \mathbf{y_1} \times \mathbf{y_2}$ that transforms the original space into a new one-dimensional space.

D	y_1	y_2	y_{num}	y_{class}
x_1	1	1	1.25	B
x_2	1	3	7.0	A
x_3	3	2	2.7	C
x_4	3	3	3.2	A
x_5	2	4	5.5	B

1. **Learn a regression model on the transformed feature space using the OLS closed form solution to predict the continuous output variable $\mathbf{y_{num}}$.**

   The first step will be to calculate the transformed feature space for each observation using the polynomial basis function:

   $$\phi(y_1, y_2) = y_1 \times y_2 \tag{1}$$

   By applying (1) to each observation, we get:

   $$\phi(x_1) = 1 \times 1 = 1 \qquad \phi(x_2) = 1 \times 3 = 3 \qquad \phi(x_3) = 3 \times 2 = 6$$
   $$\phi(x_4) = 3 \times 3 = 9 \qquad \phi(x_5) = 2 \times 4 = 8$$

   We end up with the following transformed feature space: $\phi(y_1, y_2) = [1, 3, 6, 9, 8]$

   The regression model in the transformed feature space is given by:

   $$z = w_0 + w_1 \cdot \phi(y_1, y_2)$$

   The next step is to learn a regression model using the OLS closed form solution which, for this exercise, is given by:

   $$W = (\Phi^T \Phi)^{-1} \Phi^T y_{\text{num}}$$

where Φ is the parameterization matrix and y_{num} is the output variable.

$$\Phi = \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 6 \\ 1 & 9 \\ 1 & 8 \end{bmatrix} \qquad y_{\text{num}} = \begin{bmatrix} 1.25 \\ 7.0 \\ 2.7 \\ 3.2 \\ 5.5 \end{bmatrix}$$

As suggested in the hint, we will use `numpy` to calculate the weights W:

```python
import numpy as np

M = np.array([[1, 1], [1, 3], [1, 6], [1, 9], [1, 8]])
y_num = np.array([1.25, 7.0, 2.7, 3.2, 5.5])

W = np.linalg.inv(M.T @ M) @ M.T @ y_num

W = np.round(W, 5) # round to 5 decimal digits
print(W)
```

The output of the code above is [3.31593 0.11372].
Therefore, the weights are $w_0 = 3.31593$ and $w_1 = 0.11372$.

The regression model is then given by:

$$y_{\text{num}} = 3.31593 + 0.11372 \cdot \phi(y_1, y_2)$$

2. **Repeat the previous exercise, but this time learn a Ridge regression with penalty factor $\lambda = 1$. Compare the learnt coefficients with the ones from the previous exercise and discuss how regularization affects them.**

The Ridge regression model is given by:

$$W = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y_{\text{num}}$$

where Φ is the parameterization matrix, y_{num} is the output variable (which remain the same as in the previous exercise), and λ is the penalty factor.

```python
import numpy as np

M = np.array([[1, 1], [1, 3], [1, 6], [1, 9], [1, 8]])
lambda_value = 1.0
I = np.array([[1, 0], [0, 1]])  # Identity matrix
y_num = np.array([1.25, 7.0, 2.7, 3.2, 5.5])

W = np.linalg.inv(M.T @ M + lambda_value * I) @ M.T @ y_num

W = np.round(W, 5) # round to 5 decimal digits
print(W)
```

The output of the code above is [1.81809 0.32376].
Therefore, the weights are $w_0 = 1.81809$ and $w_1 = 0.32376$.

The Ridge regression model is then given by:

$$y_{\text{num}} = 1.81809 + 0.32376 \cdot \phi(y_1, y_2)$$

In order to compare the weights obtained in the OLS and Ridge regression models, we must first normalize the data:

normalization for the OLS model:

$$w_{0\text{normalized}} = \frac{3.31593}{3.31593 + 1.81809} = 0.64587$$
$$w_{1\text{normalized}} = \frac{0.11372}{0.11372 + 0.32376} = 0.25994$$

Normalization for the Ridge Regression model:

$$w_{0\text{normalized}} = \frac{1.81809}{3.31593 + 1.81809} = 0.35413$$
$$w_{1\text{normalized}} = \frac{0.32376}{0.11372 + 0.32376} = 0.74006$$

Comparing the normalized weights obtained in the OLS and Ridge regression models, we can see that in the Ridge regression model w_0 is smaller and w_1 is larger than the corresponding OLS model weights.

The regularization term in the Ridge regression model penalizes large weights, but it allows for the weights to increase or decrease relative to one another based on the data. In this case, the regularization reduces the bias term w_0, as the model learns to rely more on the feature w_1 to explain the variance in the target variable. This reduces the need for a large bias term, helping to prevent overfitting and stabilizing the model's predictions.

These results suggest that the feature has a strong positive correlation with the target variable. As a result, w_1 increases to better capture this relationship, allowing w_0 to decrease.

3. **Given three new test observations and their corresponding output $\mathbf{x_6} = (\mathbf{2}, \mathbf{2}, \mathbf{0.7}), \mathbf{x_7} = (\mathbf{1}, \mathbf{2}, \mathbf{1.1})$, and $\mathbf{x_8} = (\mathbf{5}, \mathbf{1}, \mathbf{2.2})$, compare the train and test RMSE of the two models obtained in (1) and (2). Explain if the results go according to what is expected.**

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

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

where $\tilde{z}_i$ is the predicted value, z_i is the true value and N is the number of observations.

OLS model:

Using the values of the weights obtained in the first exercise, we can calculate the predicted values for all observations:

$$\underbrace{\Phi = \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 6 \\ 1 & 9 \\ 1 & 8 \end{bmatrix} \wedge w = \begin{bmatrix} 3.31593 \\ 0.11372 \end{bmatrix} \Rightarrow \tilde{z} = \begin{bmatrix} 3.42965 \\ 3.65709 \\ 3.99825 \\ 4.33941 \\ 4.22569 \end{bmatrix}}_{\text{train observations}} \quad \underbrace{\Phi = \begin{bmatrix} 1 & 4 \\ 1 & 2 \\ 1 & 5 \end{bmatrix} \wedge w = \begin{bmatrix} 3.31593 \\ 0.11372 \end{bmatrix} \Rightarrow \tilde{z} = \begin{bmatrix} 3.77081 \\ 3.54337 \\ 3.88453 \end{bmatrix}}_{\text{test observations}}$$

Finally, we take $z = [1.25, 7.0, 2.7, 3.2, 5.5, 0.7, 1.1, 2.2]$ along with the predicted values to calculate the two RMSE for the OLS model.

$$\text{RMSE}_{\text{train}(x_1 - x_5)} = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(z_i - \tilde{z}_i)^2} = \sqrt{\frac{1}{5} \times 25.0637837} = 2.2389$$

$$\text{RMSE}_{\text{test}(x_6 - x_8)} = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(z_i - \tilde{z}_i)^2} = \sqrt{\frac{1}{3} \times 18.23757233} = 2.4656$$

Ridge Regression model:

Repeating the exact same process for the Ridge model, we get the following predicted values:

$$\underbrace{\Phi = \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 6 \\ 1 & 9 \\ 1 & 8 \end{bmatrix} \wedge w = \begin{bmatrix} 1.81809 \\ 0.32376 \end{bmatrix} \Rightarrow \tilde{z} = \begin{bmatrix} 2.14185 \\ 2.78937 \\ 3.76065 \\ 4.73193 \\ 4.40817 \end{bmatrix}}_{\text{train observations}} \quad \underbrace{\Phi = \begin{bmatrix} 1 & 4 \\ 1 & 2 \\ 1 & 5 \end{bmatrix} \wedge w = \begin{bmatrix} 1.81809 \\ 0.32376 \end{bmatrix} \Rightarrow \tilde{z} = \begin{bmatrix} 3.11313 \\ 2.46561 \\ 3.43689 \end{bmatrix}}_{\text{test observations}}$$

We take $z = [1.25, 7.0, 2.7, 3.2, 5.5, 0.7, 1.1, 2.2]$ along with the predicted values to calculate the RMSE for the Ridge model.

$$\text{RMSE}_{\text{train}(x_1 - x_5)} = \sqrt{\frac{1}{5}\sum_{i=1}^{5}(z_i - \tilde{z}_i)^2} = \sqrt{\frac{1}{5} \times 23.18868212} = 2.1535$$

$$\text{RMSE}_{\text{test}(x_6 - x_8)} = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(z_i - \tilde{z}_i)^2} = \sqrt{\frac{1}{3} \times 9.217983941} = 1.7529$$

Comparing the RMSE values for the train and test observations, we can see that the Ridge regression model has a lower RMSE for both the train and test data compared to the OLS model. This suggests that the Ridge regression model makes more accurate predictions and fits the data better than the OLS model. The regularization term in the Ridge regression model helps to prevent overfitting and improve the generalization of the model to new data, resulting in lower prediction errors.

4. **Consider an MLP to predict the output $\mathbf{y_{class}}$ characterized by the weights**

$$W^{[1]} = \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.2 & 0.1 \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} 0.1 \\ 0 \\ 0.1 \end{bmatrix}, \quad W^{[2]} = \begin{bmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad b^{[2]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

the output activation function

$$\text{softmax}(z_c^{[\text{out}]}) = \frac{e^{z_c^{[\text{out}]}}}{\sum_{l=1}^{|C|} e^{z_c^{[\text{out}]}}}$$

, no activations on the hidden layer(s) and the cross-entropy loss:

$$\text{CE} = -\sum_{i=1}^{N} \sum_{l=1}^{|C|} t_l^{(\text{i})} \log(X_l^{[\text{out}](\text{i})}) \tag{2}$$

Consider also that the output layer of the MLP gives the predictions for the classes A, B and C in this order. Perform one stochastic gradient descent update to all the weights and biases with learning rate $\eta = 0.1$ using the training observation x_1.

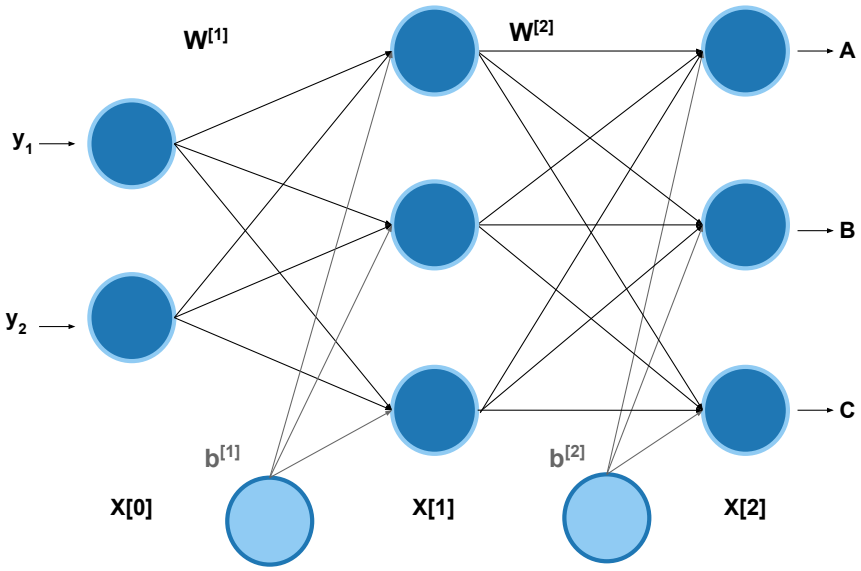


Figure 1: MLP architecture

We may also draw the corresponding scheme:

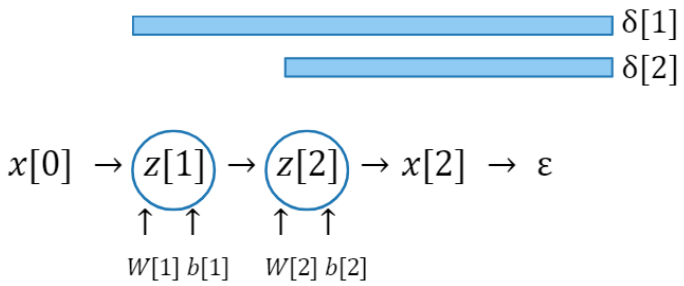


Figure 2: MLP Scheme

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

$$z_i^{[p]} = W^{[p]}\, x_i^{[p-1]} + b^{[p]} \qquad\qquad x_i^{[p]} = f\left(z_i^{[p]}\right)$$

f is the activation function for the output layer, provided in the statement.

$$f = \text{softmax}(z_c^{[\text{out}]}) = \frac{e^{z_c^{[\text{out}]}}}{\sum_{l=1}^{|C|} e^{z_c^{[\text{out}]}}}$$

In this notation, the value p is the index of the MLP layer.

We may now calculate the values of each node in the network for the training observation:

$\boxed{x_1}$

$$z^{[1]} = W^{[1]}\, x^{[0]} + b^{[1]} = \begin{bmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.2 & 0.1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \end{bmatrix}$$

$$x^{[1]} = z^{[1]} = \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \end{bmatrix}$$

$$z^{[2]} = W^{[2]}\, z^{[1]} + b^{[2]} = \begin{bmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2.7 \\ 2.3 \\ 2.0 \end{bmatrix}$$

$$x^{[2]} = f\left(z^{[2]}\right) = \begin{bmatrix} 0.461 \\ 0.309 \\ 0.229 \end{bmatrix}$$

Now, we can calculate the **backward propagation** to update the weights and biases.

Let's star by computing the derivatives for the z[i]:

$$\frac{\partial z^{[i]}(W^{[i]}, b^{[i]}, x^{[i-1]})}{\partial W^{[i]}} = x^{[i-1]}$$

$$\frac{\partial z^{[i]}(W^{[i]}, b^{[i]}, x^{[i-1]})}{\partial b^{[i]}} = 1 \qquad \frac{\partial z^{[i]}(W^{[i]}, b^{[i]}, x^{[i-1]})}{\partial x^{[i-1]}} = W^{[i]}$$

Now we calculate the deltas (for the output layer and the hidden layer):

$$\delta^{[2]} = \mathbf{x}^{[2]} - t$$

$$= \begin{pmatrix} 0.461 \\ 0.309 \\ 0.229 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.461 \\ -0.691 \\ 0.229 \end{pmatrix}$$

$$\delta^{[1]} = \left(\frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{x}^{[1]}}\right)^T \cdot \delta^{[2]} = \left(\mathbf{W}^{[2]}\right)^T \cdot \delta^{[2]}$$

$$= \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.461 \\ -0.691 \\ 0.229 \end{pmatrix} \circ \begin{pmatrix} 0.3 \\ 0.3 \\ 0.4 \end{pmatrix} = \begin{pmatrix} -0.001 \\ -0.231 \\ 0.460 \end{pmatrix}$$

Finally, we can **update the weights and biases** with the gradient descent:

$$\frac{\partial E}{\partial \mathbf{W}^{[1]}} = \delta^{[1]} \cdot \frac{\partial \mathbf{z}^{[1]}}{\partial \mathbf{W}^{[1]}} = \delta^{[1]} \cdot \left(\mathbf{x}^{[0]}\right)^T$$

$$= \begin{pmatrix} -0.001 \\ -0.231 \\ 0.460 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \end{pmatrix} = \begin{pmatrix} -0.001 & -0.001 \\ -0.231 & -0.231 \\ 0.460 & 0.460 \end{pmatrix}$$

$$\mathbf{W}^{[1]} = \mathbf{W}^{[1]} - \eta \cdot \frac{\partial E}{\partial \mathbf{W}^{[1]}} = \begin{pmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.2 & 0.1 \end{pmatrix} - 0.1 \cdot \begin{pmatrix} -0.001 & -0.001 \\ -0.231 & -0.231 \\ 0.460 & 0.460 \end{pmatrix} = \begin{pmatrix} 0.1001 & 0.1001 \\ 0.1231 & 0.2231 \\ 0.1540 & 0.0540 \end{pmatrix}$$

$$\frac{\partial E}{\partial \mathbf{b}^{[1]}} = \delta^{[1]} \cdot \frac{\partial \mathbf{z}^{[1]^T}}{\partial \mathbf{b}^{[1]}} = \delta^{[1]}$$

$$\mathbf{b}^{[1]} = \mathbf{b}^{[1]} - \eta \cdot \frac{\partial E}{\partial \mathbf{b}^{[1]}} = \begin{pmatrix} 0.1 \\ 0 \\ 0.1 \end{pmatrix} - 0.1 \cdot \begin{pmatrix} -0.001 \\ -0.231 \\ 0.460 \end{pmatrix} = \begin{pmatrix} 0.0999 \\ -0.0231 \\ 0.0540 \end{pmatrix}$$

$$\frac{\partial E}{\partial \mathbf{W}^{[2]}} = \delta^{[2]} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}} = \delta^{[2]} \cdot \left(\mathbf{z}^{[1]}\right)^T$$
$$= \begin{pmatrix} 0.461 \\ -0.691 \\ 0.229 \end{pmatrix} \cdot \begin{pmatrix} 0.3 & 0.3 & 0.4 \end{pmatrix} = \begin{pmatrix} 0.1383 & 0.1383 & 0.1844 \\ -0.2073 & -0.2073 & -0.2764 \\ 0.0687 & 0.0687 & 0.0916 \end{pmatrix}$$

$$\mathbf{W}^{[2]} = \mathbf{W}^{[2]} - \eta \cdot \frac{\partial E}{\partial \mathbf{W}^{[2]}} = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} - 0.1 \cdot \begin{pmatrix} 0.1383 & 0.1383 & 0.1844 \\ -0.2073 & -0.2073 & -0.2764 \\ 0.0687 & 0.0687 & 0.0916 \end{pmatrix}$$
$$= \begin{pmatrix} 0.9862 & 1.9862 & 1.9816 \\ 1.0207 & 2.0207 & 1.0276 \\ 0.9931 & 0.9931 & 0.9908 \end{pmatrix}$$

$$\frac{\partial E}{\partial \mathbf{b}^{[2]}} = \delta^{[2]} \cdot \frac{\partial \mathbf{z}^{[2]^T}}{\partial \mathbf{b}^{[2]}} = \delta^{[2]}$$

$$\mathbf{b}^{[2]} = \mathbf{b}^{[2]} - \eta \cdot \frac{\partial E}{\partial \mathbf{b}^{[2]}} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - 0.1 \cdot \begin{pmatrix} 0.461 \\ -0.691 \\ 0.229 \end{pmatrix} = \begin{pmatrix} 0.9539 \\ 1.0691 \\ 0.9771 \end{pmatrix}$$

Part II: Programming

Consider the parkinsons.csv dataset (available at the course's webpage), where the goal is to predict a patient's score on the Unified Parkinson's Disease Rating Scale based on various biomedical measurements. To answer question (5), average the performance of the models over 10 separate runs. In each run, use a different $80-20$ train test split by setting a `random_state = i`, with $i = 1...10$.

5. **Train a Linear Regression model, an MLP Regressor with 2 hidden layers of 10 neurons each and no activation functions, and another MLP Regressor with 2 hidden layers of 10 neurons each using ReLU activation functions. (Use `random_state=0` on the MLPs, regardless of the run). Plot a boxplot of the test MAE of each model.**

```
import matplotlib.pyplot as plt, pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.neural_network import MLPRegressor

# Read the dataset
```

```
df = pd.read_csv("./parkinsons.csv")
X, y = df.drop("target", axis=1), df["target"]

linear_mae = []

# Linear Regression
for i in range(1, 11):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=i)

    lr_model = LinearRegression()
    lr_model.fit(X_train, y_train)

    y_pred = lr_model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    linear_mae.append(mae)

mlp_no_activation_mae = []

# mlps with no activation function
for i in range(1, 11):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=i)

    mlp_no_activation = MLPRegressor(hidden_layer_sizes=(10, 10), activation='identity',
                                     random_state=0, validation_fraction=0.2)
    mlp_no_activation.fit(X_train, y_train)

    y_pred = mlp_no_activation.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    mlp_no_activation_mae.append(mae)

mlp_relu_mae = []

# mlps with ReLU activation function
for i in range(1, 11):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=i)

    mlp_relu = MLPRegressor(hidden_layer_sizes=(10, 10), activation='relu',
                            random_state=0, validation_fraction=0.2)
    mlp_relu.fit(X_train, y_train)

    y_pred = mlp_relu.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    mlp_relu_mae.append(mae)

# Data for plotting
b_plot = plt.boxplot(
    [linear_mae, mlp_no_activation_mae, mlp_relu_mae], patch_artist=True,
    labels=['Linear Regression', 'MLP No Activation', 'MLP ReLU']
)

```

```
colors = ["#1f77b4", "#E40071"]
for patch, color in zip(b_plot["boxes"], colors):
    patch.set_facecolor(color)
for median in b_plot["medians"]:
    median.set_color("black")
# Create boxplot
plt.title('Test MAE of Models')
plt.ylabel('Mean Absolute Error')
plt.grid(axis="y")
plt.show()
```

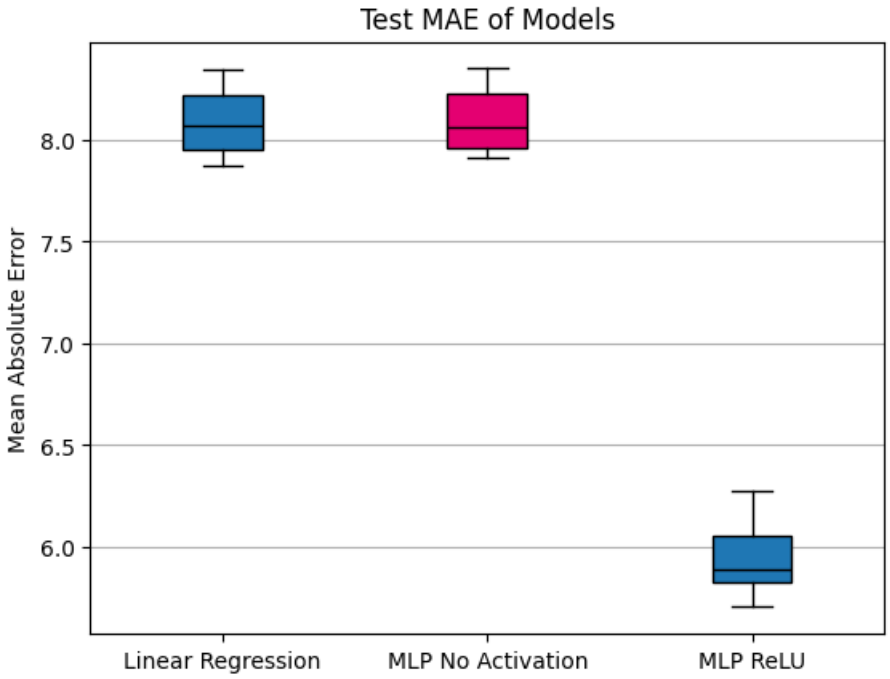


Figure 3: Boxplot of the test MAE for each model

6. **Compare a Linear Regression with a MLP with no activations, and explain the impact and the importance of using activation functions in a MLP. Support your reasoning with the results from the boxplots.**

   As we can see in the previous plot, the MLP without activation functions behaves similarly to Linear Regression, while the MLP with ReLU activation function performs better than the other two models. Linear Regression and MLP without activation functions have a high MAE, centered around 8.0. On the other hand, the MLP with ReLU activation function has a lower MAE, centered around 6.0 and this shows the power of activation functions. Activation functions enable the MLP to learn from non-linear relationships in the data. Without them, each layer is just performing a linear operation, which severely limits the network's ability to generalize and fit real-world data, especially when the underlying relationships are complex or non-linear.

7. **Using a $80-20$ train-test split with `random_state=0`, use a Grid Search to tune the hyperparameters of an MLP regressor with two hidden layers (size 10 each). The parameters to search over are: (i) L2 penalty, with the values $\{0.0001, 0.001, 0.01\}$; (ii) learning rate, with the values $\{0.001, 0.01, 0.1\}$; and (iii) batch size, with the values $\{32, 64, 128\}$. Plot the test MAE for each combination of hyperparameters, report the best combination, and discuss the trade-offs between the combinations.**

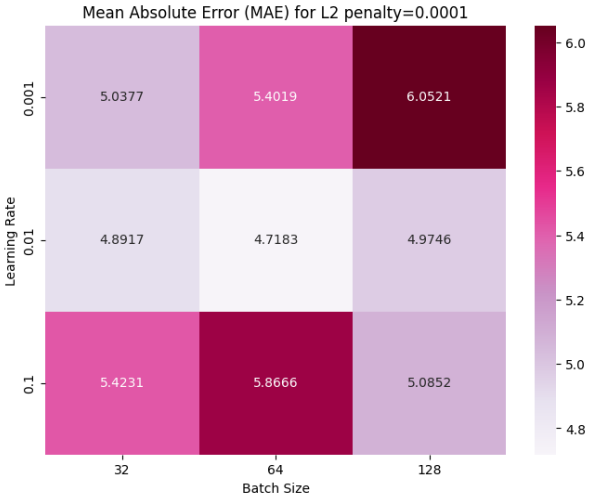
```python
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset (Assuming the file is named 'parkinsons.csv')
data = pd.read_csv('parkinsons.csv')

# Separating features and target
df = pd.read_csv("./parkinsons.csv")
X, y = df.drop("target", axis=1), df["target"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=0)

# Define the MLPRegressor with two hidden layers (size 10 each)
mlp = MLPRegressor(hidden_layer_sizes=(10, 10), random_state=0)

# Define the hyperparameters to search
param_grid = {
    'alpha': [0.0001, 0.001, 0.01],  # L2 penalty
    'learning_rate_init': [0.001, 0.01, 0.1],  # Learning rate
    'batch_size': [32, 64, 128],  # Batch size
}

# Perform grid search
grid_search = GridSearchCV(mlp, param_grid, cv=3, scoring='
    neg_mean_absolute_error')
grid_search.fit(X_train, y_train)

# Get the best parameters and best model
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

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

# Evaluate the model using Mean Absolute Error (MAE)
test_mae = mean_absolute_error(y_test, y_pred)
print(f"Test MAE: {test_mae}")
print(f"Best Hyperparameters: {best_params}")

# Visualize the test MAE for each combination of hyperparameters, grouped
```

```
    by 'alpha'
results = pd.DataFrame(grid_search.cv_results_)
results['mean_test_mae'] = -results['mean_test_score']  # Convert to
    positive MAE

# Plot a heatmap for each value of alpha
alphas = results['param_alpha'].unique()

for alpha in alphas:
    subset = results[results['param_alpha'] == alpha]
    pivot_table = subset.pivot(index='param_learning_rate_init', columns='
    param_batch_size', values='mean_test_mae')

    # Plot the heatmap
    fig, ax = plt.subplots(figsize=(8, 6))
    sns.heatmap(pivot_table, annot=True, fmt=".4f", cmap="PuRd", ax=ax)
    plt.title(f'Mean Absolute Error (MAE) for alpha={alpha}')
    plt.xlabel('Batch Size')
    plt.ylabel('Learning Rate')
    plt.show()
```

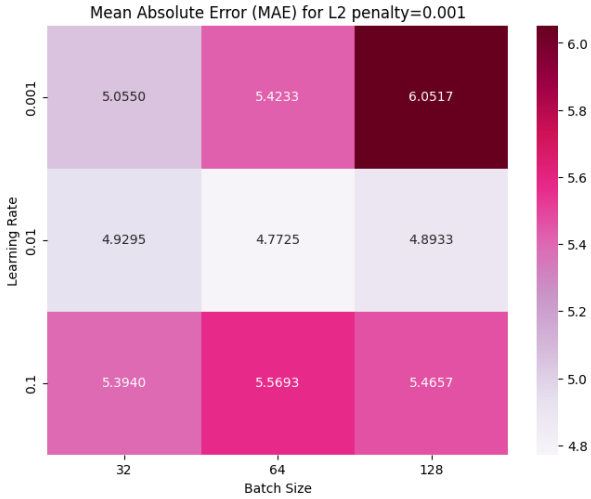
When analizing MAE values (graphics below), the smaller the MAE the better the models performance. With that being said, the best combination for heatmap is:

- Learning Rate = 0.01 and Batch Size = 64 for L2 penalty = 0.0001
- Learning Rate = 0.01 and Batch Size = 64 for L2 penalty = 0.001
- Learning Rate = 0.01 and Batch Size = 32 for L2 penalty = 0.01

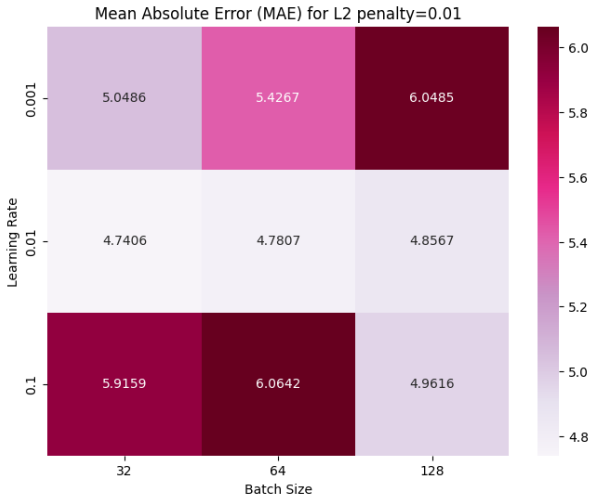
The best combination of hyperparameters for the MLP regressor, achieving the lowest Mean Absolute Error (MAE) of 4.7183, consists of an L2 penalty (alpha) of 0.0001, a learning rate of 0.01, and a batch size of 64. Lower alphas help avoid over-penalizing the model, while the 0.01 learning rate offers a good balance between convergence speed and avoiding overshooting. Batch size 64 provides the best generalization. Higher alphas and learning rates led to higher errors due to over-regularization and poor convergence.



(a) MAE plot for L2 penalty = 0.0001



(b) MAE plot for L2 penalty = 0.001



(c) MAE plot for L2 penalty = 0.01

Figure 4: MAE plot for each combination of hyperparameters